Steganography*

Leonardo Rezende Costa - RA262953**

I. Introduction

This report describes the work done related to the fourth project proposed on class. The aim of this project is to hide some content into a bitmap colored image using the least significant bits of each pixel channel. The input images have different sizes and the input texts have different length.

II. Script execution

This section describes how to execute the script and it's dependencies.

The ***code.py*** and ***uncode.py*** files have the scripts used in this work to generate the results. The ***exec.sh*** is a bash script with the execution of five tests to hide and recover the input texts into and from the images. It can be executed by ***./exec.sh*** in an environment with all the dependencies listed on subsection II-A

A. Dependencies

The dependencies to run the python3 scripts are:

- Numpy
- Opencv-python

III. Implementation

This section describes how the hiding and the recovering functions works and what was the approach to develop the script.

A. Hiding

The hiding process begins with cleaning the right plane of every pixel channel to receive the text bit later. The first approach was to wipe the plane of the entire image, that is:

$$image = image \wedge \overline{2^{plane}} \tag{1}$$

this means that every pixel channel will pas trough a bit-wise operation **and** ($\wedge$) with the complement of the number that represents the plain ($\overline{2^{plane}}$).

For example, if the objective is to store a message on the fourth (plain = 3) bit-plane, we do the operation

$$Pixel_{h,w,c} = \begin{matrix} b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{matrix} \wedge \tag{2}$$

resulting on

$$Pixel_{h,w,c} = b_7 \ b_6 \ b_5 \ b_4 \ 0 \ b_2 \ b_1 \ b_0 \tag{3}$$

where *h*, *w* and *c* are the address referent to *row*, *column* and *channel*, respectively, and the zero represent the wiped b_3 referent to the fourth plane that we want to clean.

Then, it was realized that we lose the whole plain, and not just the channels we need to store the content, so a mask-based approach was developed. On this approach, we just wipe the amount of channels that are equal to $8 * message\_size$. The Fig. 1 shows the building process for the mask.

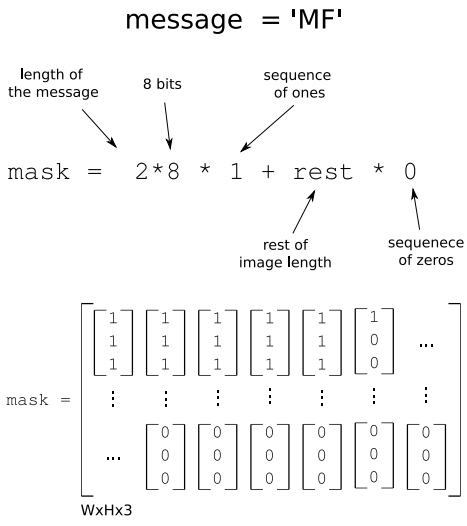

Fig. 1: Mask building example to wipe just the necessary amount of pixel channels to store the input text on the input image. In this example, only 16 bits referent to the two letters were used.

After building the mask, we multiply it by 2^{plane} to shift bit to the left and then make a bit-wise operation between the image and the $\overline{mask}$ (the complement of the mask). The *numpy* **bitwise_and** relates the correspondent cells between the arrays when they have the same shape; this way the mask serves to wipe the desired bits.

After wiping, we insert a final character \0 a the end of the input text and then convert it into a into a 2D list **bs** (bit_sequence) where each row represents a sequence of bits for every character in the text. The Fig. 2 shows a representation for this list.

*MO443 report - Introdução ao processamento de imagens digitais

**Instituto de Computação - Mestrado, Universidade Estadual de Campinas

$$BS = \begin{bmatrix} [b_{07}\ b_{06}\ b_{05}\ b_{04}\ b_{03}\ b_{02}\ b_{01}\ b_{00}] \\ [b_{17}\ b_{16}\ b_{15}\ b_{14}\ b_{13}\ b_{12}\ b_{11}\ b_{10}] \\ \cdots \\ [b_{n7}\ b_{n6}\ b_{n5}\ b_{n4}\ b_{n3}\ b_{n2}\ b_{n1}\ b_{n0}] \end{bmatrix}$$

Fig. 2: List of characters converted into a sequence of bits. The letter **n** represents the number of characters.

After converting each character into a binary sequence, we reshaped the array as the image shape. This way, we have the bits positioned to match the pixel channels of the image. Fig. 3 shows a representation of the array reshaped.

$$BS = \begin{bmatrix} \begin{bmatrix} b_{07} \\ b_{06} \\ b_{05} \end{bmatrix} \begin{bmatrix} b_{04} \\ b_{03} \\ b_{02} \end{bmatrix} \begin{bmatrix} b_{01} \\ b_{00} \\ b_{17} \end{bmatrix} \begin{bmatrix} b_{16} \\ b_{15} \\ b_{14} \end{bmatrix} \begin{bmatrix} b_{13} \\ b_{12} \\ b_{11} \end{bmatrix} \cdots \\ \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ \cdots \begin{bmatrix} b_{n7} \\ b_{n6} \\ b_{n5} \end{bmatrix} \begin{bmatrix} b_{n4} \\ b_{n3} \\ b_{n2} \end{bmatrix} \begin{bmatrix} b_{n1} \\ b_{n0} \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{bmatrix}$$

Fig. 3: List of sequence of bits reshaped into the image's shape. The last bits are zero to show that when the input text don't need the whole bit plane, zeros are added into the list to fulfill the new structure.

The array is then shifted to the left to put the bit in the right position of the plane.

With the text bit sequence prepared and the correspondent pixel channel bits wiped, an **or** operation between the *BS* and the image is made to store the values and then the image is saved.

B. Recovering

To recover the text hidden in the image, we simply do a bit-wise **and** between the image channels and the value 2^{plane} to wipe the rest of the image and then shift the bit to the right, so the image is transformed into an array almost equal to the **BS** described at III-A. The array is reshaped into Nx8 to return the values to 8-bit sequences and then every row is converted back to a character and concatenated into the output text.

IV. Results

The Fig. 4 shows the images[1] used to test the scripts.

[1]All those images was get at <https://www.ic.unicamp.br/~helio/imagens_coloridas/>, accessed in 31/10/2019

Fig. 4: Original images used as input on the tests.

Five tests were made to check the scripts. The first used a lorem ipsum generator [1] to create text with length equals to $\frac{3}{8}$pixels of the image and test the full capacity of it. Fig. 5 shows the result image on this test. The text has 98304 ascii characters and they were stored in the least significant bit (LSB) of each pixel channel. We can notice that the steganography isn't perceptible to the eye when we compare the result to the original image. Even without the original image, we could not say that an message was hidden because the noise don't really change the intensity levels of R, G or B.

Fig. 5: Result for storing 98304 chars into the plain 0 of the pepers.png image. Left: original image. Right: the image with the text.

The second test stored a lyric with 1438 characters into the *baboon* input and the result is showed at Fig. 6.

Fig. 6: Result for storing 1438 characters into the plain 0 of the baboon.png image. Left: original image. Right: the image with the text.

In the third test an smaller text were stored in the *monalisa* input image and the result is showed at Fig. 7.

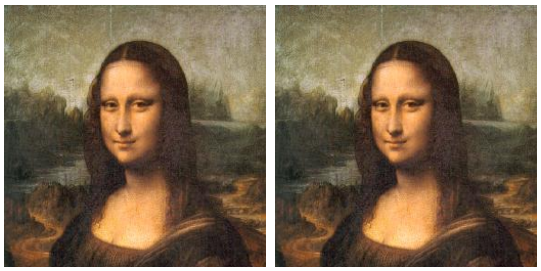

Fig. 7: Result for storing 571 characters into the plain 2 of the monalisa.png image. Left: original image. Right: the image with the text.

The fourth test was an smaller text were stored in the *watch* input image and the result is showed at Fig. 8. This test aims to produce a result to be compared to the fifth test.

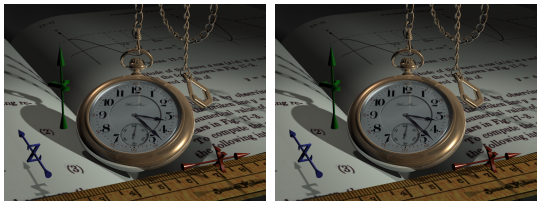

Fig. 8: Result for storing 571 characters into the plain 1 of the watch.png image. Left: original image. Right: the image with the text.

The fifth test was the same text from the first test stored in the *watch* input image and the result is showed at Fig. 9. In the first test, the text used the whole space of the plain to be stored. In this one we used the plain 6, and this means that we used more significant bits to store the message. In this plain, the noise in the result is perceptible at a human eye, even without the original image to compare.

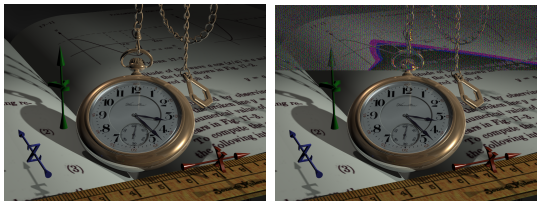

Fig. 9: Result for storing 98304 characters into the plain 6 of the watch.png image. Left: original image. Right: the image with the text.

Also, all those tests are using the mask approach to change just the necessary bits in the image, causing a more clean result. In Fig. 10 we can see the result of the fifth test without this approach.

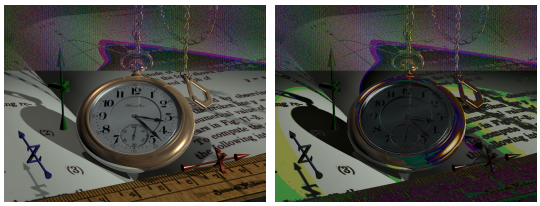

Fig. 10: Result for the fifth test without the wiping-mask approach. Left image: result of changing just the necessary bits to store the message. Right image: result of wiping every bit of the plain 6 before storing the input text0.

A. Bit plains

In order to compare the results, the bit plains 0,1,2 and 7 were saved as image to see how the bits were distributed. Fig. 11 shows the four plains of the first result. We can notice that the top-left image stored the message and, as the message took the whole plain, the plain doesn't looks like the original image. We can see that the plain 1 and 2 have a little information about the black spots on the original image, but the plain 7 represents the original image in a much better way, as expected, considering that the most significant bit would store critical information about the image.

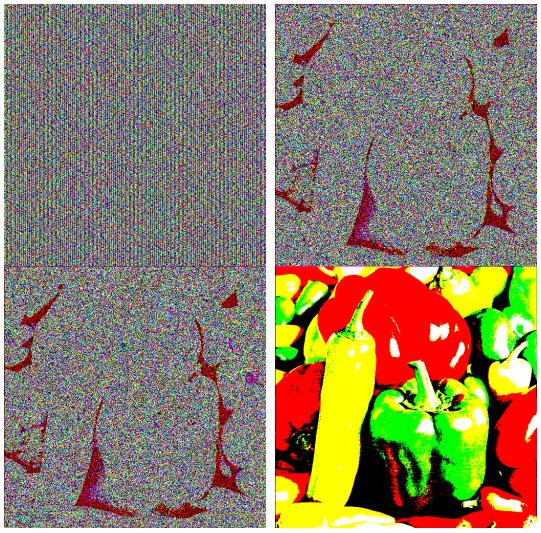

Fig. 11: Top left to bottom right: plain 0, plain 1, plain 2 and plain 7 of the first test.

The next plain we can observe is the one related to the third test, where not all plain were necessary to store the image. Fig. 12 shows the second plane with and without the wiping-mask approach. We can notice that the left image is almost black. This is because without the mask we just put zero on the whole plain and then store the message. In this case, the message only took a small fraction of the plane (the noisy lines in the beginning) and the rest was set to black.

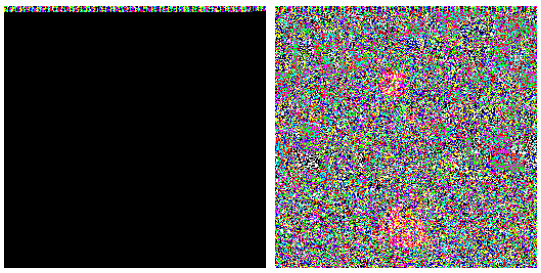

Fig. 12: Plain 2 of the third test. Left image: result without the wiping mask. Right image: result with the wiping mask, that maintains the original pixel values that wasn't necessary to store the message

The Fig. 13 shows the bit plains splitted by channel. This test was made with the longest input text on the bit-plane 2. In this case, the image has low contrast and the colours are more dark than white. This means that the MSB doesn't represent as much information as in the other images. The plain 2 determines a lot of variation on the colors, so when we store the image, the output coded version shows a strange pattern and the somebody would probably note the difference. Fig. 14 shows the result and we can see that the upper part of it suffers more visual changes as expected. We can also notice that the plain 2 plots on Fig. 13 give us much of the scene forms, different of the same plain 2 on Fig. 11 that give us a slightly idea of the peppers.

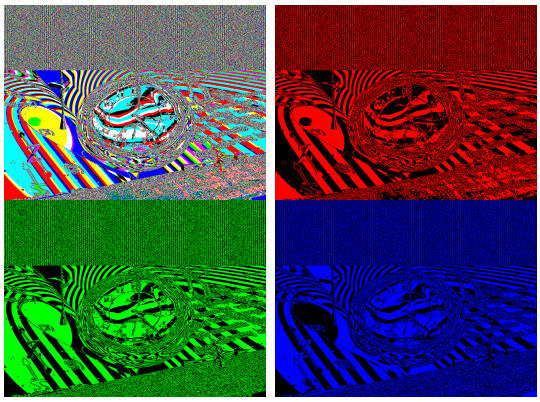

Fig. 13: Result of hiding a 98304 length text into the watch.png image on bit-plane 2. Top-left to bottom-right: all channels, red channel, green channel and blue channel

Fig. 14: Output image with the message referent to Fig. 13. We can notice the difference on the first third part of the image, specially on the paper colour above the watch

V. Conclusion

As the name suggests, the least significant bits doesn't influences the data value when it's composed by various bits. In the image processing context, a pixel containing 8 bits per channel means that the LSB represents 1/256 of the value of the intensity and using this bit to store another types of data results in an image almost equals to the original. In this project we explored this characteristic and saw that even using the third bit (third bit plane) resulted in an image almost equal to the input one, but it depends on the image's contrast. That's because when the contrast is low and the histogram shows the intensities on the darkest side, LSB are probably more important than brighter images with higher contrast. Another detail is that the character words doesn't need to be 8-bits, as the ascii just uses 7. This means that the scripts could be implemented by using just sequences of 7 bits stored in the image, resulting in a larger space to hide a content.

References

[1] Lorem ipsum, https://pt.wikipedia.org/wiki/Lorem_ipsum, Wikipedia, Wikimedia Foundation, 2019, October