Project 2

Isabela Telles Furtado Doswaldo - RA: 170012

MC920 - Introduction of Digital Image Processing
University of Campinas

1 Introduction

The goal of the project is to implement an algorithm of steganography in digital images.

Steganography is the practice of concealing a message into an image. The message could be a text, an image, a video, or a file. Steganography can be applied, for instance, in a confidential communication or, most common, in digital watermarking, as the producers embed digital watermarks into the image to foil pirates that try to beat the system.

The programs developed in this project encode and decode images using Python, along with the following libraries: Imageio, Matplotlib and Numpy.

The images used for testing are available in http://www.ic.unicamp.br/~helio/imagens_coloridas_png/ and the results presented here are outputs of the program when it was runned with the image *peppers.png*.

2 Methodology

The method used in this project was the least significant bit, which consists in modifying one of the three least significant bits of each pixel of the image, so the message is stored in these modified bits. Next subsections provide a description of how each program works.

2.1 Program: Encode

This program is able to encode a message into an image.

2.1.1 Execute

There are four arguments necessary to run the program:

1. Input image: path to the image file in which the message will be embedded
2. Input text: path to the text file with the message to be hidden
3. Bit plane: bit plane indicating in which layer of the image the message will be hidden
4. Output image: path to the image file with embedded message

See an example of usage to execute the program when the input image is `peppers.png`, the input text is `text.txt`, the bit plane is 0 and the image encoded will be `peppers_encoded.png`:

```
$ python3.5 codificar.py peppers.png text.txt 0 peppers_encoded.png
```

2.1.2 Read Image

To load the image from a file into a 3D numpy array, the Imageio library's function *imageio.imread()* was used. Since the function returns the image data as-is, it was necessary to use the function *astype()* of the Numpy library to cast the numpy array to `uint8` type.

2.1.3 Convert Text to Binary

First, the text was read using the *with* statement along with the *open()* function, so the file was properly closed after its suite finished, even if an exception was raised at some point. Then, with the file opened and the file object created, the latter called the function *read()* that returned data as a string.

In order to convert the string to binary, it was necessary to iterate through it and convert each character to an integer representing its Unicode code point using the function *ord()*. Each integer was formatted to a string of 8 binary digits with the *format()* function and the result strings were joined into one using the *join()* function.

2.1.4 Encode Message into Image

To encode the message into the image, the 3D numpy array representing the image was iterated, as well as the string, which had the binary code. Each bit of the binary code was set into the bit plane, specified as argument, of each pixel of the image. So, for example, a pixel with RGB = (192, 32, 43) could be represented by bit arrays RBG = (11111101, 00100000, 00101011) that are nothing but an 8-bit representation of the value in each channel.

To set the bit into the bit plane of each bit array it was necessary to do some bit-wise operations. First of all, a mask was created shifting the bits of 1 to the left by 0, 1 or 2 places, depending on the bit plane specified. If the bit that was going to be set were 1, a logical OR was done between the mask and the bit array, so the bit plane's bit of the bit array would be overwritten by the bit in question. Otherwise, a logical AND was done between the complement of the mask and the bit array, so bit 0 could overwrite the bit plane's bit.

In Figure 1 and 2, you can compare both images, the original and the encoded one. As it was expected, the difference is not perceptible to human vision.

Figure 1: Original Image

Figure 2: Encoded Image

2.2 Program: Decode

This program is able to decode a message of an image.

2.2.1 Execute

There are three arguments necessary to run the program:

1. Output image: path to the image file with embedded message

2. Input text: path to the text file with the message to be hidden

3. Bit plane: bit plane indicating in which layer of the image the message was hidden

See an example of usage to execute the program when the output image is `peppers_encoded.png`, the input text is `output.txt` and the bit plane is 0:

```
$ python3.5 decodificar.py peppers.png 0 output.txt
```

2.2.2 Read Image

To read the encoded image, the same procedure described in section 2.1.2 was done.

2.2.3 Decode Message of Image

In order to decode the message of the image, it was necessary to iterate through the 3D numpy array representing the image.

To extract the bit of the bit plane, the bits of the value in each channel were shifted to the right by 0, 1 or 2 places, depending on the bit plane specified. Then, the remainder of the shifted number divided by 2 was, indeed, the bit that needed to be extracted.

Each bit extracted was added into a string named binary code, and when the 8th bit was extracted, the *chr()* function was used to convert the binary code to the corresponding character. So, each character obtained was added into a string named text.

2.2.4 Create Bit Planes 0, 1, 2 and 7

While the message was recovered, the iteration through the 3D numpy array was also used to create the bit planes 0, 1, 2 and 7. Those were used to determine, indeed, whether the image had a message encoded or not.

Before the iteration, four ndarrays of Numpy library were created, with the same shape of the original image and the same type as well, `uint8`. Then, during the iteration, the bit plane's bit was extracted using the same procedure described in section 2.2.3. The bits extracted would create a binary image of zeros and ones, which would damage the visualization of the bit planes' images. So, the bits were multiplied by 255, which allowed a better visualization of a message's existence, as you can see in Figures 3, 4, 5 and 6. The third figure shows that the bit plan 0 is certainly the one where the message was encoded, because the details of the shadows of the image, showed in Figure 1 and in Figure 4, that is, the pixels with highest frequencies of the image were hidden by a certain pattern. In Figures 4 and 5 those pixels are evident.

2.2.5 Write Text

The recovered text was written in a file with the path specified as argument using the *with* statement along with the *open()* function, such as mentioned in section 2.1.3. The file object created called the *write()* function with the text string as argument.

Below you can see a piece of the text encoded into the image of example:

```
Low-key, no pressure
just hang with me and my weather
Low-key, no pressure
just hang with me and my weather

Rose-colored boy
I hear you making all that noise
About the world you want to see
And oh, I'm so annoyed
'Cause I just killed off what was left of the optimist in me
```

```
Hearts are breaking
Wars are raging on
And I have taken my glasses off
You got me nervous
I'm right at the end of my rope
A half empty girl
Don't make me laugh, I'll choke
```

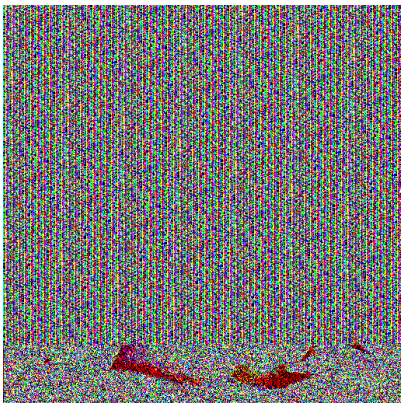

Figure 3: Bit Plan 0

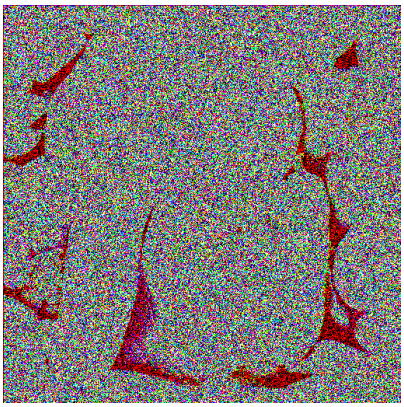

Figure 4: Bit Plan 1

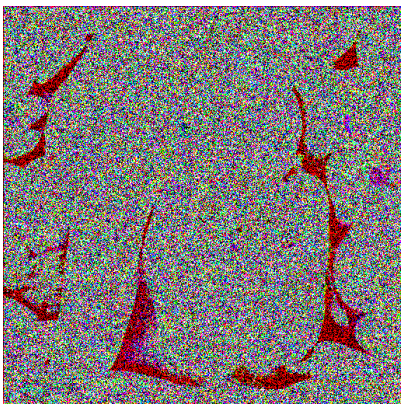

Figure 5: Bit Plan 2

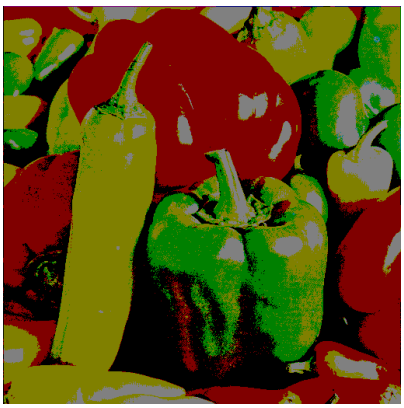

Figure 6: Bit Plan 7