Aix-Marseille University

Computational and Mathematical biology

Stenography: Least Significant Bit

Author:
VASILEVA Svetlana

Supervisors:
Julien LEFEVRE, Jérémie PERRIN

Programming & Algorithms

January 3, 2023

Student Name: VASILEVA Svetlana
Student ID: v22028163

Programming and Algorithms
Project report

1 Introduction

Notes from author. This project topic was chosen because of a deep interest in image analytics. Undoubtedly, at present this niche has been completely filled by ML methods, but it is important for me to probe the old-school motives of this branch. To be honest, the main part of this project turned out to be its research side, and it seemed interesting to me to delve into the issues of weak points of stenography, to consider how in the prime time of this method, people tried to circumvent it. But let's start step by step.

Steganography (literally from the Greek "secret writing") is the science of transmitting hidden data (stegocommunications) in other open data (stegocontainers). Unlike cryptography, which hides the contents of a given message, shorthand hides the very fact of its existence. As a rule, the message will look like something else, for example, like an image, an article, a shopping list, a letter.

In our case, we will hide our message in the image, so let's take a closer look at this method.

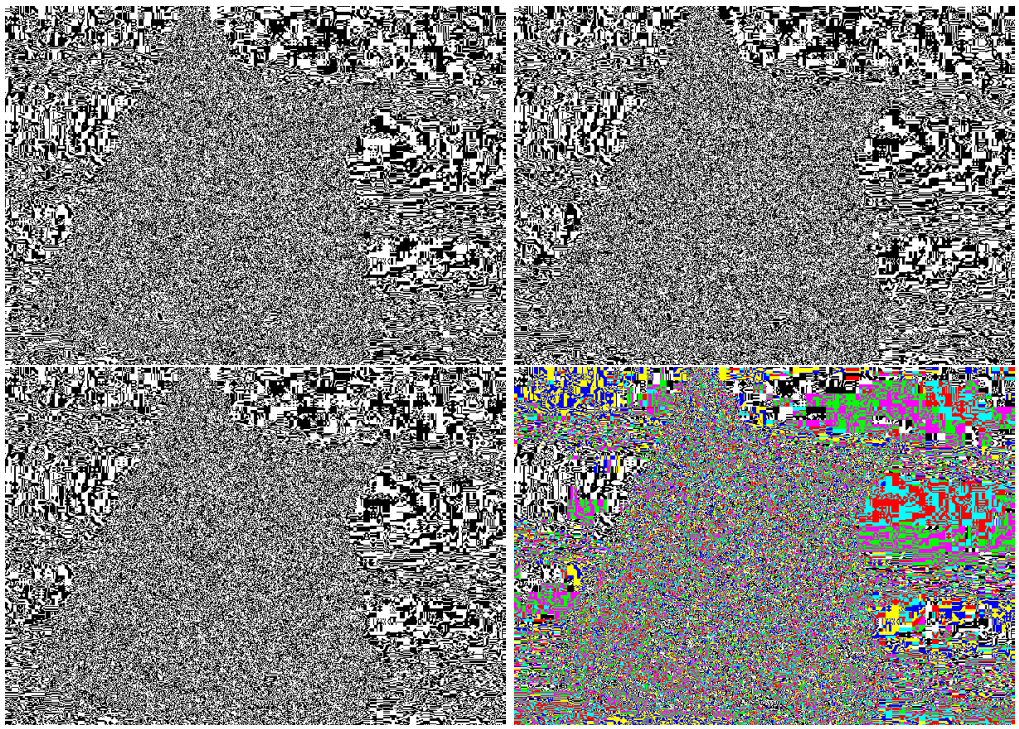


Figure 1: Separated LSB channels: 1 - red, 2 - blue, 3 - green, 4 - all

1.1 Least significant bit

The most popular color model is RGB, where the color is represented as three components: red, green and blue. Each component is encoded in the classical version using 8 bits, that can take a

value from 0 to 255. This is where the least significant bit is hiding. It is important to understand that one RGB color accounts for as many as three such bits. To present them more clearly, let's do a couple of small manipulations. With a picture of a cat, of course.

Let's take a picture with a cat in PNG format. We'll split it into three channels and take the least significant bit in each channel. Let's create three new images, where each pixel stands for LSB. Zero is a white pixel, and one is black, respectively. Below are the images that are obtained. But usually the image is found in "assembled form". To represent the LSB of three components in one image, it is enough to replace the component in a pixel, where LSB is one, with 255, and in the opposite case, replace it with 0(Figure 1[4]).

Let's imagine that our field of work is the last image. Let's take this as a stream of bits that we can modify. All we need is our message itself, which we want to encode and the picture where we will do it (in this example, our cat in LSB interpretation). The next step is to present our secret message in the form of bits and sequentially write in place of the existing ones.

1.2 Why does it work?

Let's try to compare empty and full containers:

(a) Empty container (b) Almost full container

It's almost impossible to recognise the difference with the first glance, with the second and even with the third as well. We can try to distinguish just two pixels (0, 0, 0) and (1, 1, 1), that have difference only in LSB in each component.

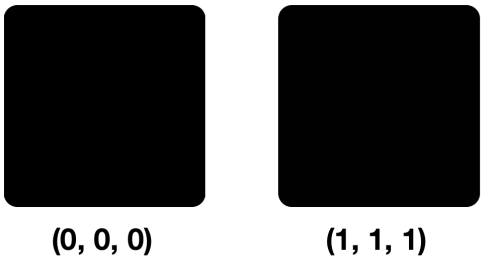


Figure 3: Adjacent colors in the RGB palette

The fact is that our eye can distinguish about 10 million colors, and the brain can only distinguish about 150. The RGB model also contains 16,777,216 colors. You can check how many you can distinguish with this picture:

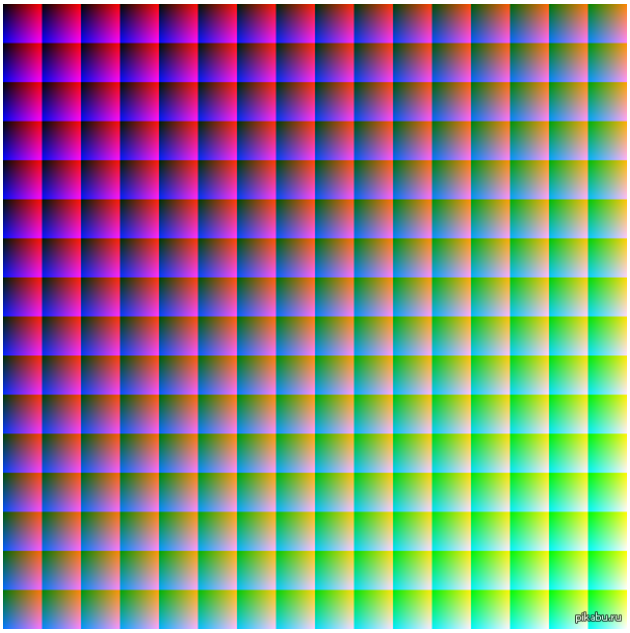


Figure 4: All colors from the RGB palette

2 Code

There is the classic beginning of code part - libraries describtion.

```
import cv2
import docopt
import numpy as np
```

Package **cv2** is designed for processing and analysis of computer vision. Feature detection, image processing, video analysis, camera calibration, computational photography and machine learning are on the list. The tools of this package allow us to freely climb inside the image and dispose of its bits (mask them as we please). **Docopt** helps us to equip a user-friendly interface, so to speak, to maneuver the front-end side. And **numpy** is the classic tools heap for comfortable work with matrices.

We will crate our own class, take an image as an input, process the main parameters of it: height, width, channels. There you can see the scheme of masking of our bits (inspired by internet references).

```
class Project():
    # Initialisation of our work field: reading the characteristics of input image: width
        , height, channels
    def __init__(self, im):
        self.image = im
        self.height, self.width, self.non_bi_channels = im.shape
        self.size = self.width * self.height

```

```
        # The schemes of masking - OR and AND, NB: we need to remove the first used bit!
        self.maskONEValues = [1 << 0, 1 << 1, 1 << 2, 1 << 3, 1 << 4, 1 << 5, 1 << 6, 1
            << 7]
        self.maskONE = self.maskONEValues.pop(0)
        self.maskZEROValues = [255 - (1 << i) for i in range(8)]
        self.maskZERO = self.maskZEROValues.pop(0)

        # Current position
        self.current_width = 0
        self.current_height = 0
        self.current_channel = 0
```

The next function is the main in shuffling our bites. There we manipulate the bits to hide our info (also in bits). We 'climb inside' the architecture of the image.

```
    def climb_inside(self, bits):

        for c in bits:  # Iterate over all bits: look inside 3D matrix
            val = list(self.image[self.current_height, self.current_width])
            # Fix in the image the info about the bit: if it is set - change, if not -
                remain the bit from the input pic
            # Depending on the status we chose the certain mask
            if int(c):
                val[self.current_channel] = int(val[self.current_channel]) | self.maskONE
            else:
                val[self.current_channel] = int(val[self.current_channel]) & self.
                    maskZERO

            # Update image
            self.image[self.current_height, self.current_width] = tuple(val)

            # The function described below - move pointer to the next space
            self.next_step()
```

Since we process each bit separately, we need to keep our finger on the pulse - check all the time whether we have gone beyond the boundaries of the drawing and if we still have space to hide more bits of our secret info. Therefore, we will write a separate function to move to the next bit with all the checks.

```
    def next_step(self):
        # Firstly - check if we are still inside the workflow (3 checks for each
            parameter)
        if self.current_channel == self.non_bi_channels - 1:
            self.current_channel = 0
            if self.current_width == self.width - 1:
                self.current_width = 0
                if self.current_height == self.height - 1:
                    self.current_height = 0
                    # Check if we are on the final step
                    if self.maskONE == 128:
                        raise SteganographyException("No available slot remaining (image
                            filled)")
                    else:  # else go to next bitmask
                        self.maskONE = self.maskONEValues.pop(0)
                        self.maskZERO = self.maskZEROValues.pop(0)
```

```
                else:
                    self.current_height += 1
            else:
                self.current_width += 1
        else:
            self.current_channel += 1
```

Additional functions for comfortable work: **read_bit, read_byte, read_bits** - for work with certain bit, byte or any number of bits respectively. **byte_value, binary_value** - for returning int or byte form of the numbers.

And there is the launching functions for both of our processing - *encoding and decoding.*

```
    def encode_text(self, txt):
        length = len(txt)
        bin_length = self.binary_value(length, 16)  # Generates 4 byte binary value of
            the length of the secret msg
        self.climb_inside(bin_length)  # Put text length coded on 4 bytes
        for char in txt:  # And put all the chars
            c = ord(char)
            self.climb_inside(self.byte_value(c))  # update the image
        return self.image

    def decode_text(self):
        ls = self.read_bits(16)  # Read the text size in bytes
        l = int(ls, 2)  # Returns decimal value ls
        i = 0
        hidden_text = ""
        while i < l:  # Read all bytes of the text
            tmp = self.read_byte()  # So one byte
            i += 1
            hidden_text += chr(int(tmp, 2))  # Every chars concatenated to str
        return hidden_text
```

2.1 Check of the code

Our start container is the image of penguin drown by Artificial Intelligence DALL-E.

Figure 5: Empty container

Let's imagine that we are agents on a top-secret mission with a modest budget and want to encrypt our message. The program will meet us with an intuitive interface.

```
Which operation would you like to perform?
             1.Encode Text
             2.Decode Text
             3.Exit Program

1
Please describe the path to the image:
pin.png

Which message we should hide?
Hello, Mr. President. I have an information.

Creating encrypted image.

Put the destination and the name of file with secret info:
pin_mes.png

Saving image in destination.
Encryption complete.
The encrypted file is available at pin_mes.png
```

Figure 6: Screenshot of the output

Double check if our changing is perceptible by eye:

(a) Empty container

(b) Container with the message

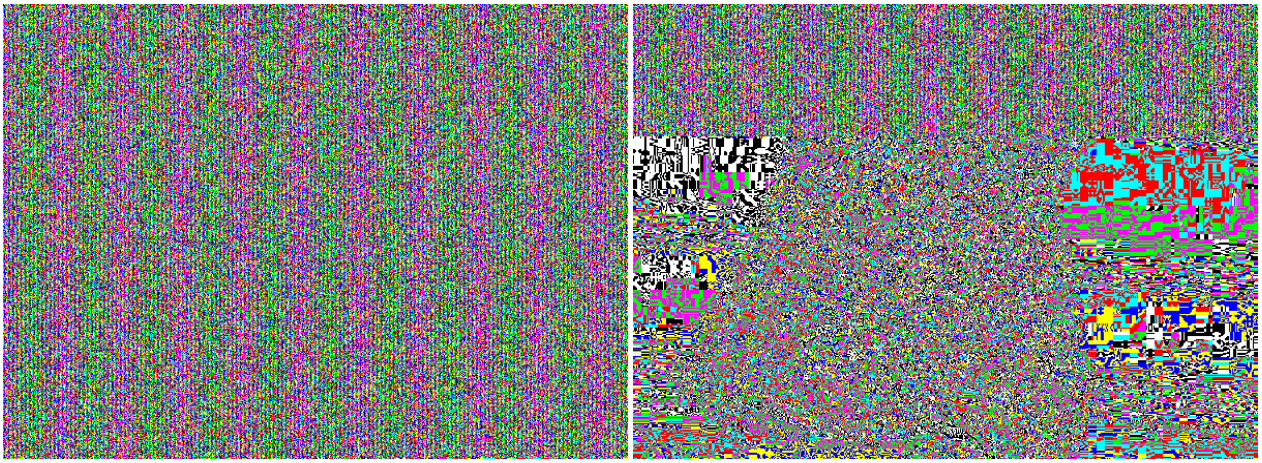
All is okay, so let's try decode our message.

```
Which operation would you like to perform?
             1.Encode Text
             2.Decode Text
             3.Exit Program

2
Enter working directory of source image:
pin_mes.png

Text message obtained from decrypting image is: Hello, Mr. President. I have an information.
```

Figure 8: Screenshot of the output

3 Weak spots of stenography

In this part of the report, I would like to go back to our kitten and talk more about the subtleties of recognizing hidden messages. As we have already understood, it is almost impossible to determine the fact of message transmission from the whole container and you need to look at the LSB picture. Here is a cat filled with fully transmitted information and a cat filled only by a third.



(a) Fully filled container

(b) Filled by a third container

Here it is worth making a small digression and talking about the size. What is a 30% cat? The size of the cat is 603x433 pixels. 30% of this size is equal to 78459 pixels. Each pixel contains 3 bits of information. A total of 78459*3 = 235377 bits or slightly less than 30 kilobytes fits in 30% of the cat. And a whole cat will fit about 100 kilobytes.

When we change a picture in any way, we change its statistical properties, so we need to find a way to fix these changes. Andreas Wesfield and Andreas Pfitzmann from the University of Dresden started using **chi-squared** for this in their work "Attacks on Steganographic Systems". The attack is carried out on one channel, and subsequently the average for all three channels is used to evaluate the statistical properties of the assembled image.

So, the Chi-square attack is based on the assumption that the probability of simultaneous appearance of neighboring (different by the least significant bit) colors (pair of values) in an unfilled stegocontainer is extremely small. In other words, the number of pixels of two adjacent colors is significantly different for an empty container. All we need to do is count the number of pixels of each color and apply a couple of formulas.

Let h be an array in the i-th place containing the number of pixels of the i-th color in the image under study. Then:

1. The measured frequency of color appearance $i = 2k$: $n_k = h[2k], k \in [0, 127]$

2. Theoretically expected frequency of color appearance: $i = 2k$: $n_k^* = \frac{h[2k]+h[2k+1]}{2}, k \in [0, 127]$

We need to keep in mind that we are working with neighboring colors, with colors (numbers) that differ only by the least significant bit. They go in pairs sequentially. If the number of pixels of 2k and 2k+1 colors will vary greatly, then the measured frequency and the theoretically expected one will differ, which is normal for an unfilled stegocontainer.

We can interpret it in python language:

```
for i in range(0, len(color) // 2):
    expected.append(((color[2 * i] + color[2 * i + 1]) / 2))
    observed.append(color[2 * i])
```

Where *color* - quantity of pixels of the color i in the image, $i \in [0, 255]$

The chi-squared criterion for the number of degrees of freedom k-1 is calculated as follows (k is the number of different colors, i.e. 256):

$$\chi^2_{k-1} = \sum_{i=1}^{k} \frac{(n_k - n_k^*)^2}{n_k^*}$$

And finally, P is the probability that the distributions n_i and n_i^* are equal under these conditions (the probability that we have a filled stegocontainer in front of us). It is calculated by integrating the smoothness function:

$$P = 1 - \frac{1}{2^{\frac{k-1}{2}}\Gamma\left(\frac{k-1}{2}\right)} \int_0^{\chi^2_{k-1}} e^{-x/2} x^{(k-1)/2-1} dx$$

It is most effective to apply chi-squared not to the whole image, but only to its parts, for example, to strings. If the calculated probability for a string is greater than 0.5, then we will paint over the string in the original image in red. If less, then green. For a cat with 30% fullness, the picture will look quite precise:

Figure 10: Percentage of changed LSBs

4 Conclusion

Summing up, we can give a couple of tips to a beginner in shorthand writing:

1. Embed a message in random bytes (one of the ways to modify the code)
2. Minimize the amount of information being implemented
3. Select images with noisy LSB content
4. Use unique images

The acquired masking skills can be used in the future not only when processing images, but also addresses, videos and texts. In the future, it would be interesting for me to understand the application of Machine Learning methods for such purposes.

5 References

There is the list of used links:

1. https://habr.com/ru/post/480428/
2. https://habr.com/ru/company/ruvds/blog/664252/
3. https://habr.com/ru/post/121989/
4. https://github.com/RobinDavid/LSB-Steganography
5. https://github.com/chewett/lsb_stenography_python