Chapter 1 - LSB Matching

By: Himanshu Sharma

This chapter discusses the implementation of LSB Matching algorithm which was proposed in 2006/03/13 by Jarno Mielikainen for publication in IEEE Signal Processing Magazine.

IEEE Signal Processing Letters: Year 2006, Volume: 13, Issue: 5
IEEE Journals and Magazines
https://ieeexplore.ieee.org/document/1618698/

Introduction

LSB Matching is a technique of hiding text data into an image. Unlike LSB encoding, LSB Matching is a different algorithm which actually manipulates pixel by simple arithmetic instead of directly replacing the LSB of pixel with the message bits. This algorithm allows hiding the same amount of data with fewer changes in the pixel values of cover image. One point to be noted is that this algorithm cannot be used for pixels which have pure white value, i.e., 255. The reason for this is that this algorithm might return the new pixel value as 256 which saturates to 255 again in any digital image. Hence, our message bit will get lost. So, care must be taken while choosing a cover image for the text data.

I will be using Python for implementing LSB Matching algorithm. MATLAB could be used as well. The following python packages will be used.

1. PIL.Image - To load image and its pixels in main memory.
2. numpy - To handle matrix and array mathematics.
3. scipy.misc - To save the modified image efficiently.

LSB Matching Algorithm

In this section we will be discussing the algorithm which is used to embedd the text message in the image. I will be manipulating the blue channel of the pixels. The algorithm starts with taking *two* consecutive pixels (actually their blue channels) at a time and at the same time taking two consecutive message bits. The algorithm then calculates the output pixels by operating on these two pixels and two message bits. Before we move further, we define a function called *Binary Function* which is the main component of this algorithm. The following defines a binary function which operates on two integers l and n.

$$binary\_function(l, n) = LSB\left(\left\lfloor \frac{l}{2} \right\rfloor + n\right) \tag{1}$$

Where, LSB function returns the least significant bit of the binary equivalent of $\left\lfloor \frac{l}{2} \right\rfloor + n$. In general, $LSB(n)$ returns the least significant bit of binary equivalent of n. Example, $LSB(5) = 1$, $LSB(2) = 0$, etc.

Let us now come to the algorithm itself. We take two input pixels which are consecutive, say, x_i and x_{i+1} and two consecutive message bits, say, m_i and m_{i+1}. Note that I said message bits, i.e., m_i and m_{i+1} are either 1 or 0.

Algorithm 1: LSB Matching

```
if $m_i = LSB(x_i)$ then
    if $m_{i+1} \neq binary\_function(x_i, x_{i+1})$ then
        $y_{i+1} = x_{i+1} + 1$
    end if
    else
        $y_{i+1} = x_{i+1}$
    end if
    $y_i = x_i$
end if
else
    if $m_{i+1} = binary\_function(x_i - 1, x_{i+1})$ then
        $y_i = x_i - 1$
    end if
    else
        $y_i = x_i + 1$
    end if
    $y_{i+1} = x_{i+1}$
end if
```

To get back the message bits, i.e., m_i and m_{i+1} from the modified pixel values y_i and y_{i+1}, we do the following process.

$$m_i = LSB(y_i) \tag{2}$$

$$m_{i+1} = binary\_function(y_i, y_{i+1}) \tag{3}$$

In the next section, we will see a demo.

Demo

We will take the following image as input and then we will try to store the following message: *Let's do steganography.*.

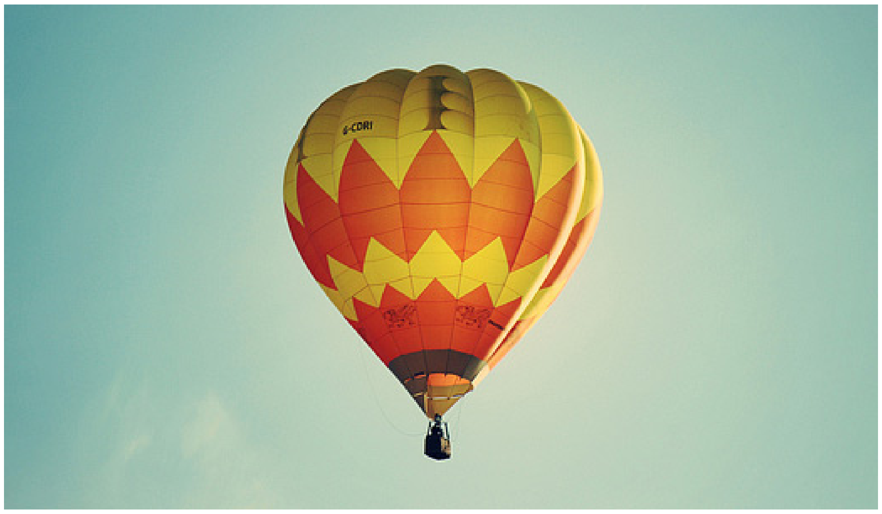

Figure 1: Cover Image

Now, in the python interpreter, type the following;

```
>>>image_embedding('inputs/balloon.png','‘Let's do steganography.'')
'Key: 184'
>>>
```

The output image after this command is;

Figure 2: Stego Image

Do you see any difference? No, right! This is the power of steganography. Our text has been embedded in this image. This is not encryption. This is hiding the text data in the image. To get back the text data, type following in the python interpreter.

```
>>> expose_message('out.png',184)
''Let's do steganography."
>>>
```

Where, out.png is the stego-image and balloon.png is the cover image. As you can see, we got our text message back on the interpreter. Note the significance of number 184. Its actually the number of pixels that the program should cover to get the message bits. If the user enters a number greater than the key (in expose_message function) provided while embedding the text in the image, the message can still be revealed but there would be extra garbage string appended to it as shown below.

```
>>> expose_message('out.png',300)
''Let's do steganography.\x9e\xf0\xab\x86\r\t\xbbP?\xbf\xbb\xd8\x16\xa9\x05"
>>>
```

If, however, the user enters a key less than the value provided to him, then the message would be incomplete.

```
>>> expose_message('out.png',140)
''Let's do steganog\x07"
>>>
```

Hence, as long as you provide the correct key or a key having value greater than the required key, you will be able to recover the data.

Conclusion

The LSB Matching algorithm is working completely fine in python and is very efficient in storing the data in the image.

✧✧✧