Digital Image Processing Project - Final Report

Group 14
Douglas Seiti Kodama - USP Number: 9277131
Leonardo de Souza Lemes - USP Number: 8941126

1. Project Objective

Our project's main objective is to hide text within colored images using a few similar approaches, generating a JPEG image, which is a method of lossy compression for digital images (so we expect to lose some of the text's data), and a PNG (with lossless compression, thus no loss) image. Then we are going to compare both images and check how the text was affected by the compression, and calculate how different it is from the original one.

2. Input Images

Our project does not require any specific image, since any image with the necessary size is eligible to become a container image, so we don't need to obtain images from any specific place, for this reason, we will use some sample images obtained from Pexels (www.pexels.com) and Flaticon (www.flaticon.com). The container images will have several distinct sizes and they will all be JPEG or PNG colored images. Some examples are listed below.

The text that is going to be hidden will be obtained from Lipsum (lipsum.com), an example is listed below.

> *"Lorem ipsum dolor sit amet."*

3. Steps Description

First step of the process is to load the container colored image on OpenCV, giving us a 3 channels matrix with pixel intensity values for each channel. The input message to be hidden will be converted to a vector with each character's respective ASCII code.

Then divide the image into the maximum amount of regions of 8x8 pixels (because the image's size might not be a multiple of 8). Each region is able to hold up to 8 characters, if you consider all 3 channels of a region, it is possible to store 24 characters in there.
Knowing the amount of regions you can get on this image, check if you can hide the text within it. The number of 8x8 blocks that fit in an image of width W and height H are: floor(W/8)*floor(H/8).
Each character is going to be hidden in a single line of the region (represented by the red rectangles), check the image below for better understanding.

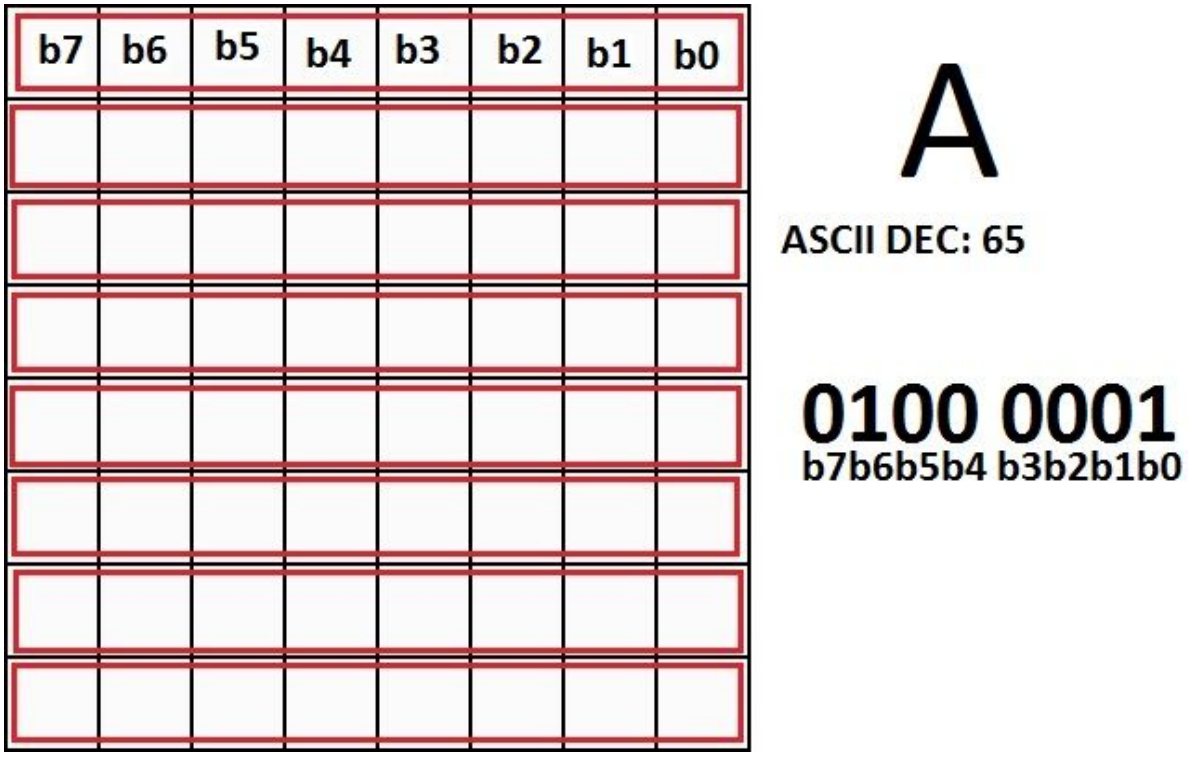


Image representation of one 8x8 block division made in the image

Each character is going to be hidden by performing small, and almost irrelevant, changes to the pixel's intensity value. To hide a single character (8 pixels are needed), replace the least significant bit (or the bit before, depending on your method) of each pixel with the character's bits. By doing so, each pixel intensity value will be increased or decreased at most 4 units, which is quite a small value given that the range of intensities vary from 0 to 255. We'll also show how hiding on more significant bits affects the image, changing the pixel intensity values by a larger amount.

After hiding the text in the image, we obtain two images: one of them is a JPEG compressed (of varying degrees) image and the other is a PNG image. We will then compare both images and check for 2 aspects:

1. how much of the original message was lost (by comparing the JPEG and PNG images)

2. how different the original image is from the container image (by comparing the JPEG and original images)

With these aspects, it is possible to calculate a percentage of lost characters after compression. After performing these calculations on some methods (varying the bit that is going to be replaced on the pixel's intensity value) on many JPEG compression levels, one can easily check the reliabilities of each method, making it easier to choose the best method to use. As we expected, we got harsh results, losing almost all content of the message using this naive technique, so we performed some minor tweaks on the algorithm to try to improve results, these tweaks are going to be explained below.

4. Results

Given the lossy nature of the JPEG compression, simply hiding the message`s bits in the least significant bits of each pixel of the image results in great loss of the message information. Using more significant bits of the pixels give better results, but it also makes visible changes to the final image.

However, even using the most significant bit of the pixel does not give satisfactory results (around 60% of message loss.

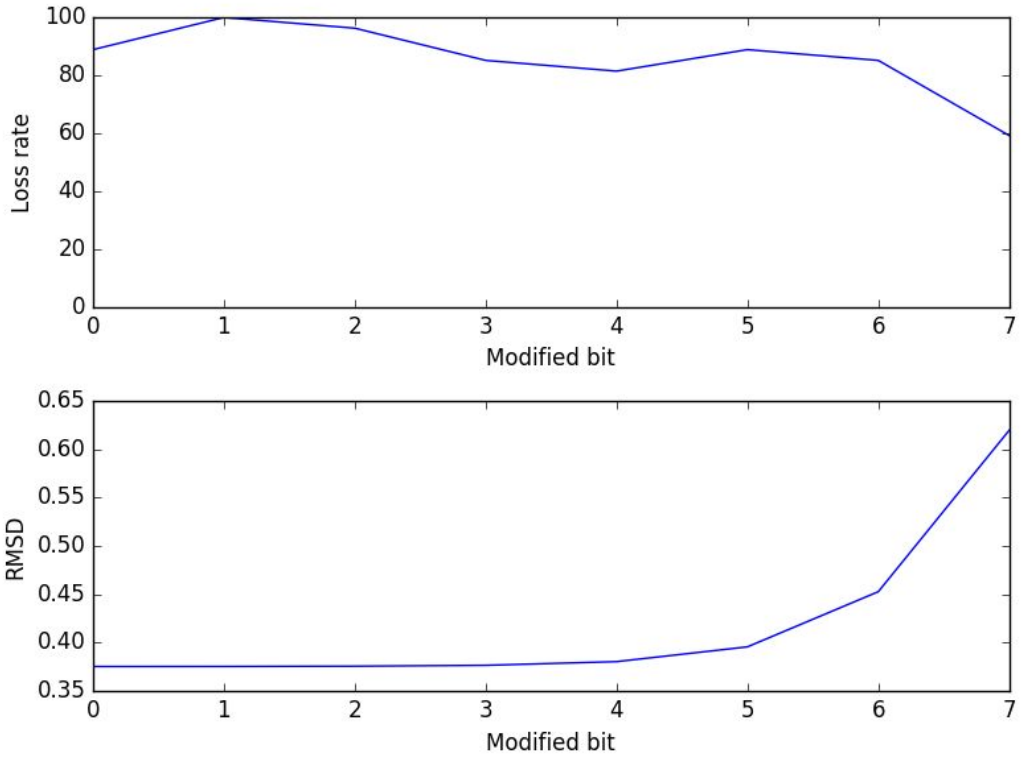
The method we used to have better results was to include redundancy in the cover image. One 8x8 block can store up to 8 different characters of the message. First we chose to fill one block with 8 copies of the same character, in one channel. For example, if the first message character was "A", the red channel of the first 8x8 block will be filled with "A" and then to recover that character, we extracted all the characters on the red channel and used the most frequent one. Using this method gave better results than simply hiding one character per line, but we still found a big message loss. To have even better results, we increased redundancy, storing now copies of a single character all over its region (which means, filling the 3 channels of a single 8x8 region with the same character). With this new strategy we managed to get 0% loss on some messages using the third least significant bit, which is nice because the image suffered minor alterations.

On the partial report, we had the idea of mapping 64 characters (from 'a' to 'z', 'A' to 'Z', ' ' and '.') to a 6 bits representation. As each region allows us to store 8 characters (64 bits), we had 8 characters (now with 48 bits), we could use 2 extra bits for each character as verifier bits, so for each character we'd have 8 bits, 6 to define the character and 2 verifiers. The idea was to use these verifiers to check the recovered message's integrity. But, given our first results, we had a quite big loss rate on each message, and these bits were useless, since we already knew we'd have losses, and that's all they would tell us. In addition, even those bits were susceptible to error, making this strategy rather useless (though, it would be useful if we had a guarantee that these bits wouldn't change) and only making us spend more space. By adding the redundancy factor, explained above, we could reduce the loss rate to about 0%, so we decided it would be better to let go of those verifier bits, instead of copying each useless

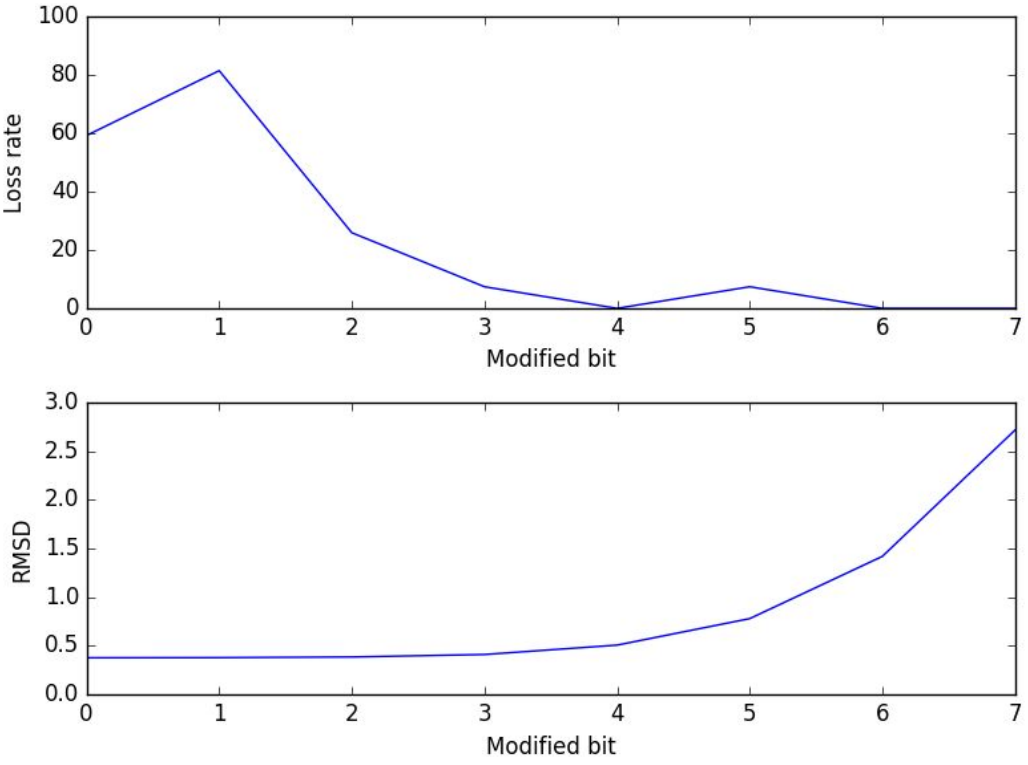
verifier bit 24 times, and map the whole ASCII table, now using 8 bits for each character and being able to store every possible character.
Here are some examples of our results, you can see more detailed information on our demo program.

For this instance, we are going to hide the message: “Lorem ipsum dolor sit amet.” on the image:

We hid the message in this image using each of the available bits (from the least significant, to the most significant), firstly, using no redundancy with a JPEG quality rate of 100%. Look at the plot.

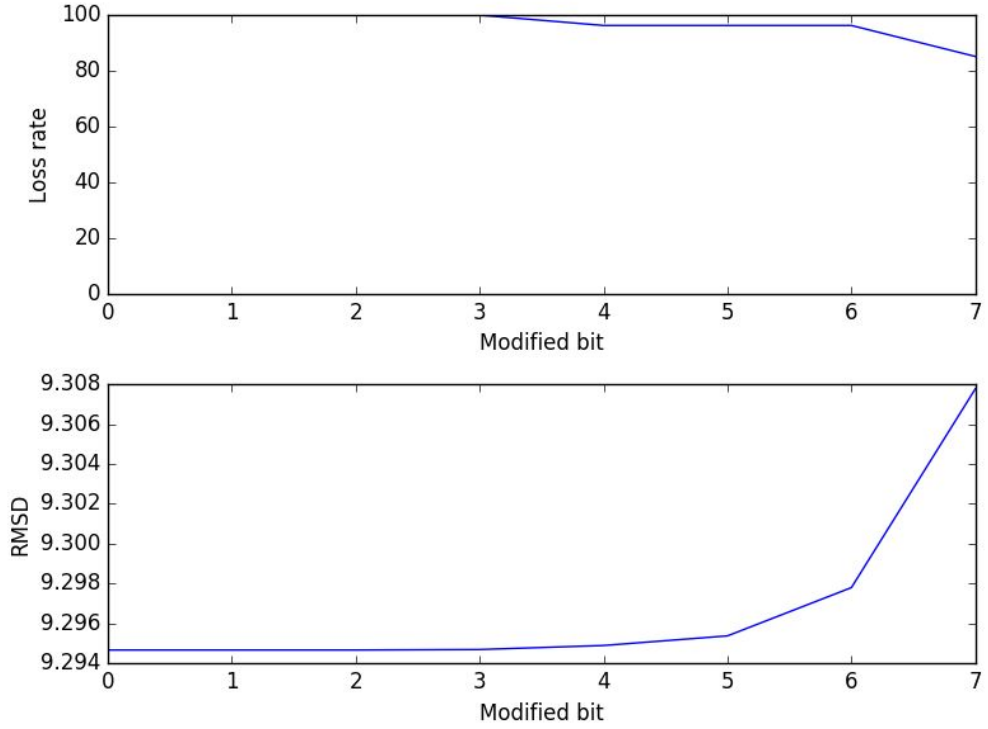


There are some bad results, even using the most significant bit, we have a 60% loss rate of the message. Now, using redundancy, with these same other aspects.



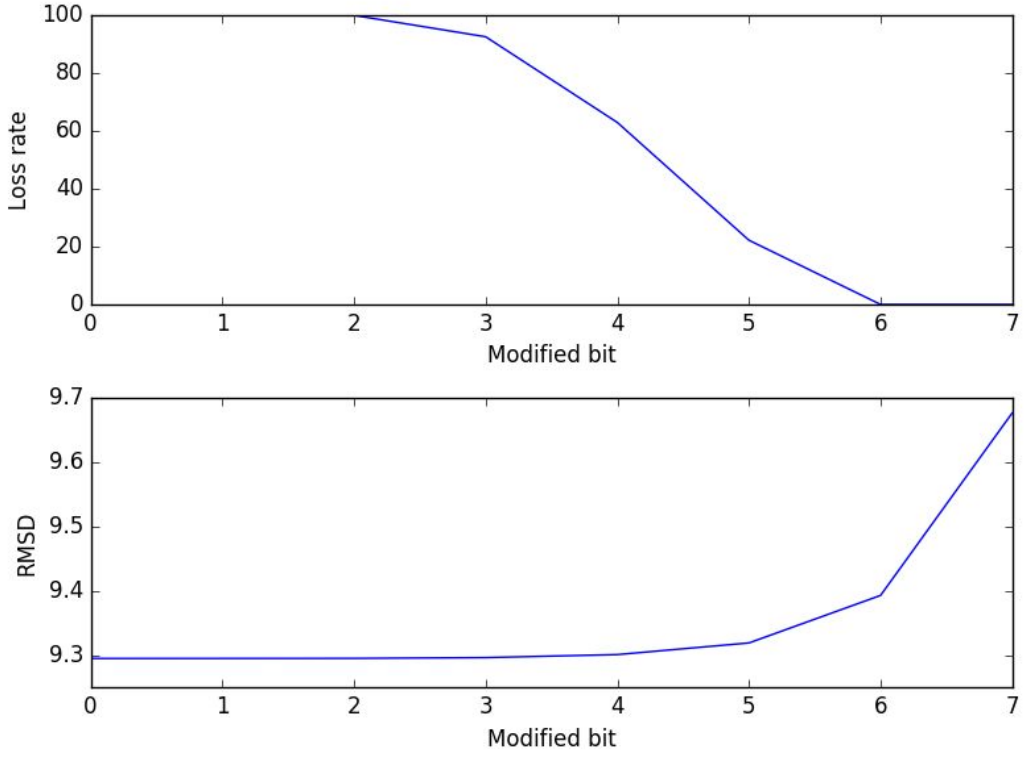
We can see that using bit 4, we hardly make any change on the image, and we get a really good loss rate.

Now we will use a JPEG quality rate of 50% and no redundancy.



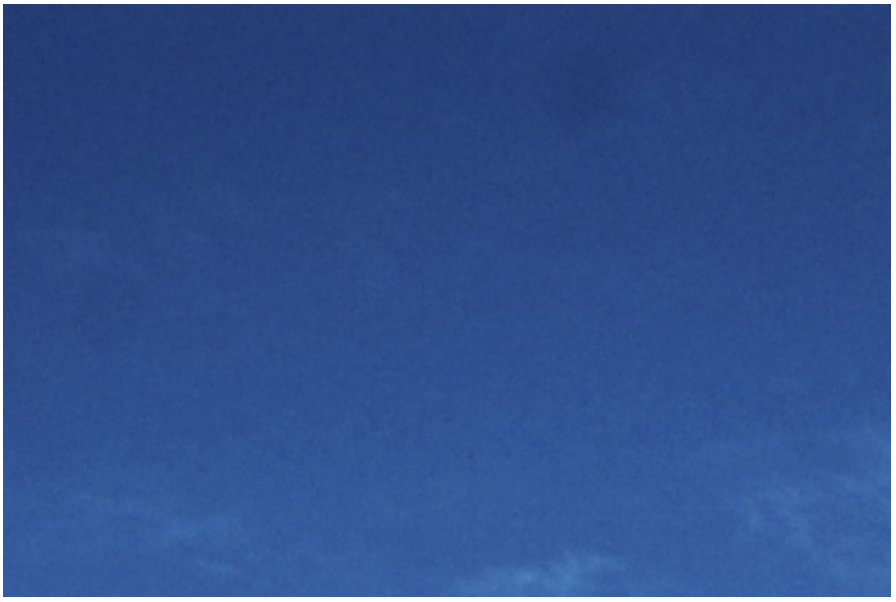
Disastrous, as you can see, we lose almost all of the message even using the most significant bit.

Now, using redundancy.



We could improve the results, but notice how hard it is to achieve an optimal method now, since JPEG quality rate is set to 50%, we lose many more bits on the compression, so we need to use almost the most significant bit to have a low loss rate, and this leads to a noticeable different image.

This is the output image, using JPEG compression, redundancy and the bit 0 you can hardly see any difference (zoomed in).



But in this case, using JPEG compression, redundancy and the bit 7, we can clearly see the difference (zoomed in).