Computational Vision Lab 01

Johannes Heidecke

October 9, 2016

Exercise 1:

The three gray scale images are created in Matlab as simple matrices containing zeros and ones. These matrices are then concatenated along the third dimension to form the three color channels of the RGB image. See figure 1.

Figure 1: Results of Exercise 1

Exercise 2:

The image *chairs.jpg* consists of the three different color channels of the RGB spectrum: red, green and blue. Pixel that have a high portion of red color (e.g. the pillows on the chairs) have a very high value in the red color channel. Looking at the greyscale image of this channel, these pixels appear almost white. The very red pillows appear black in the green and blue channel, since their color does not contain a lot of green and blue. Colors that have a almost equal distribution of the three channels are perceived as the range from white, to gray, to black. The image contains mainly white and gray colors, for that reason the three channels look very much alike, as can be seen in figure 2.

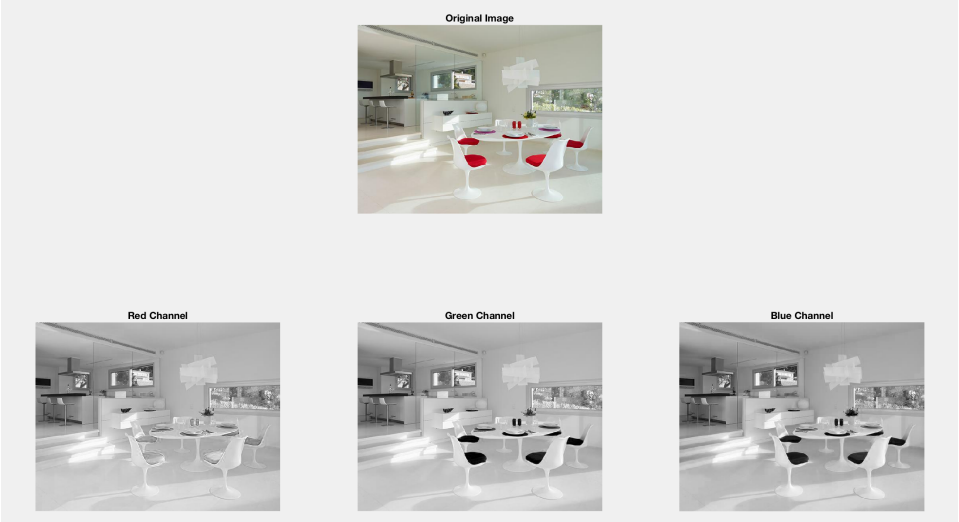

Figure 2: Original Image and its three RGB channels

It is possible to interchange the different channels with each other. Since there are three channels, 6 different permutations are possible, with one of them being the normal RGB order. The results of those permutations can be seen in figure 3.

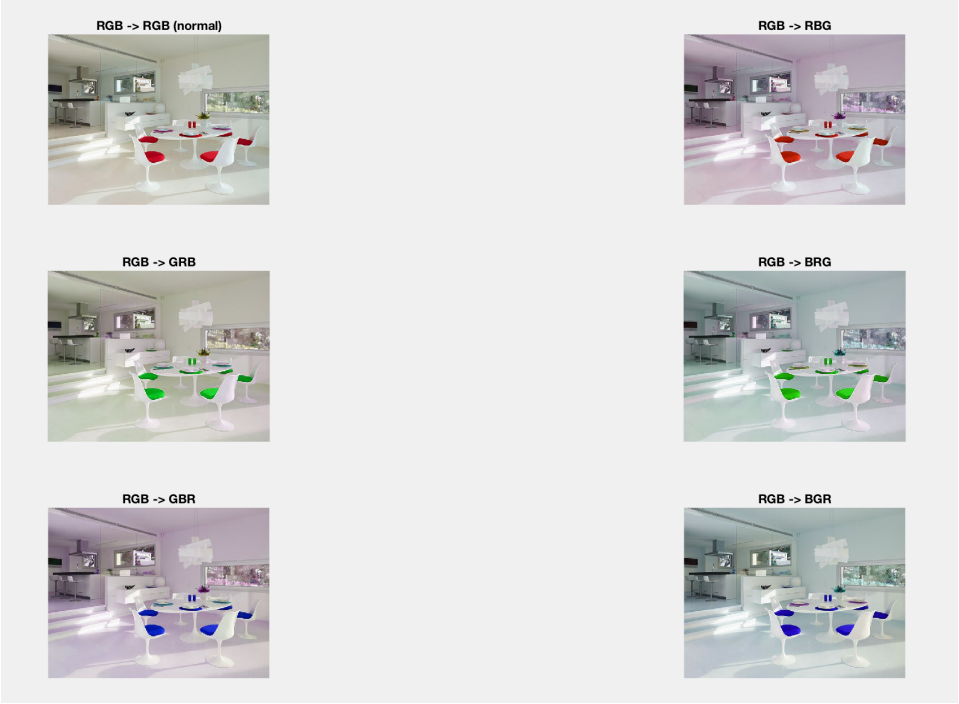

Figure 3: Interchanging the RGB channels

It is also possible to completely turn off single channels by multiplying all values of a channel by zero. The resulting images are colored in cyan, magenta, and yellow - exactly the colors that are being used in the subtractive color model of printing. The results are shown in figure 4.

Figure 4: Deactivating single RGB channels

Exercise 3:

Resizing an image to a smaller size reduces not only the size but also the details of the image. This can be seen in figure 5. While a rescaling to 0.5 times the original size still shows most of the details of this particular image, after rescaling it to 0.01 size almost all details are lost.

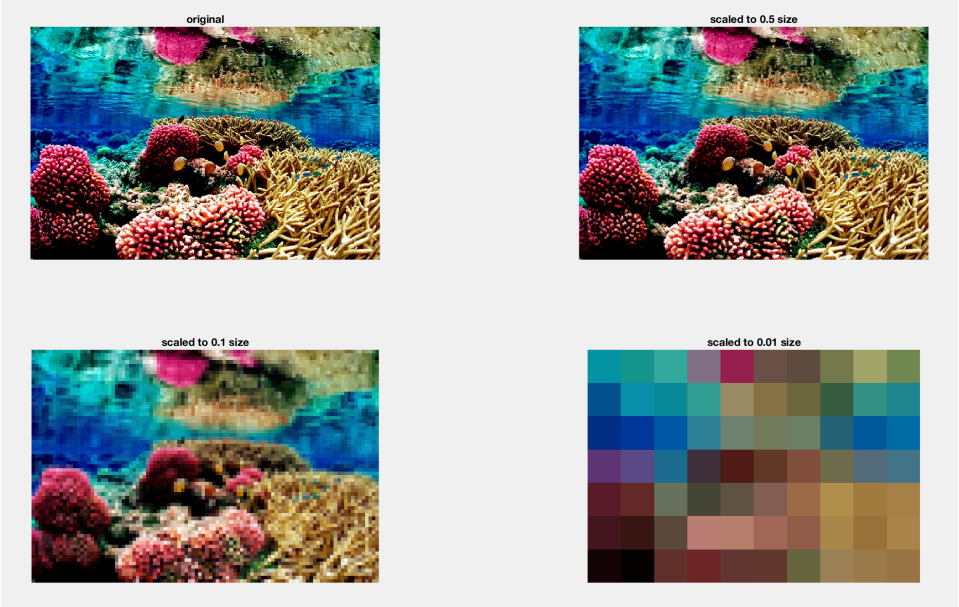

Figure 5: Resizing the original image to smaller sizes

This loss of details after resizing can also be observed in the histogram of the three color channels. In figure 6 the three histograms of the RGB-channels of the original image are shown in the left column. The same histograms are shown on the right, after reducing the size to 10% of the original size. It can clearly be observed that after resizing, the channels' values are less smoothly distributed and are concentrated on a smaller number of distinct values. This is an effect of combining the values of several pixels into one in order to allow a smaller size.

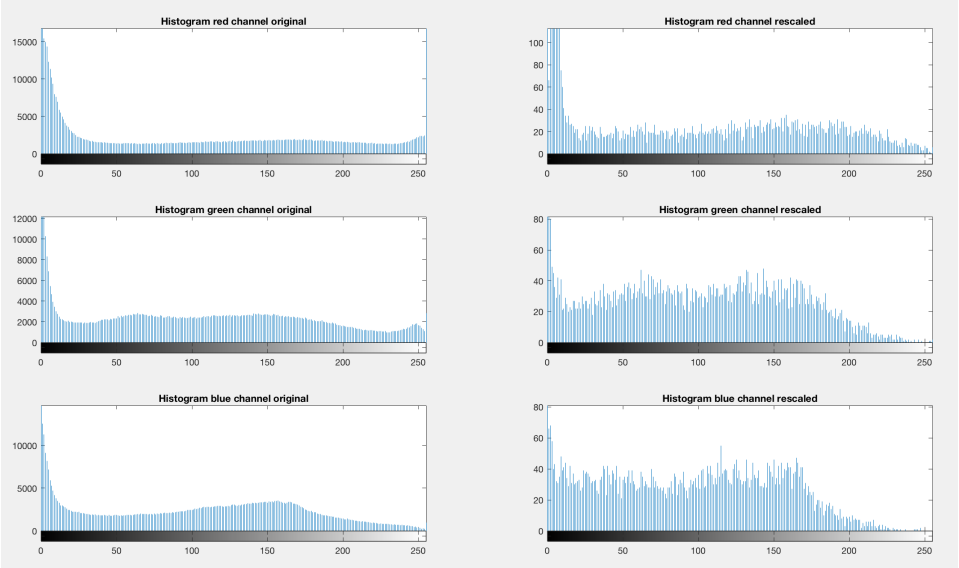

Figure 6: Change of the RGB histograms after resizing to 0.1 original size

It is possible to return the smaller image back to its original size. As figure 7 shows, the restored image does still not show the same amount of details as the original image. The details that were lost in the small image in the middle can not be re-computed. The only improvement is, that the restored image looks less pixelated, i.e. no single pixels are visible from a certain distance.

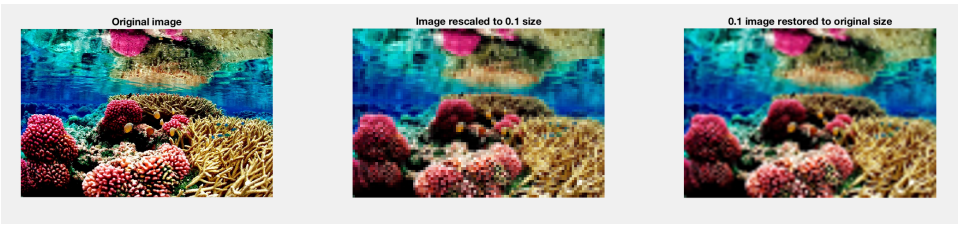

Figure 7: Resizing to 0.1 size and then restoring to original size

This effect can be observed better when only looking at a smaller area of the image, as in figure 8. This figure shows, that when resizing to 0.1 of original size, most details get lost and the image looks very pixelated.

When restoring from the 0.1 image back to original size, the pixelated look disappears, but the former details can not be recovered.

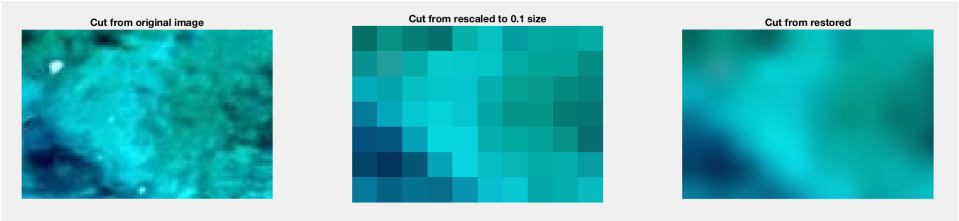

Figure 8: Resizing to 0.1 size and then restoring to original size

As an alternative for removing image details different smoothing filters can be applied. One option is to apply unweighted moving averages that simply add all values in the neighborhood of a pixel and then normalize the sum by dividing through the number of added pixels. In this case, the size and shape of the area that is chosen to calculate the average has a big impact on the outcome, as can be seen in figures 9 and 10.

In figure 9 both a horizontal filter of size 1 x 50 pixel and a vertical filter of size 50 x 1 is applied to all pixels in the image. The former results in an effect that looks like the camera was moved vertically while taking the picture, the latter like it was moved vertically. While both filters will remove some kind of noise in the image, their results still look very different.

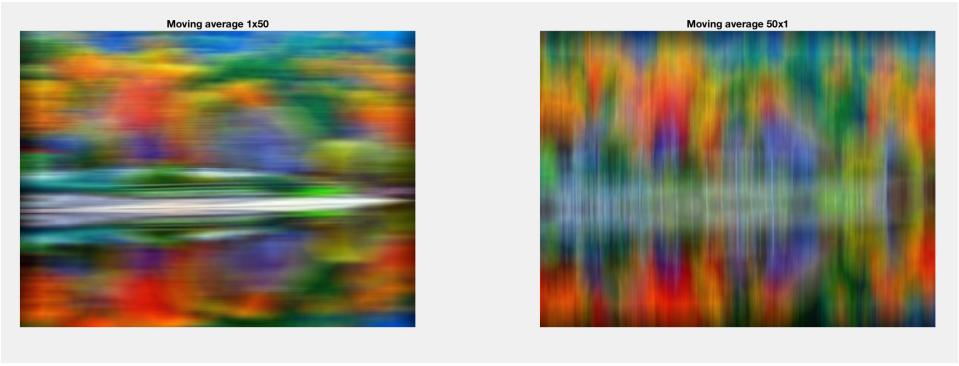

Figure 9: Horizontal and vertical unweighted moving average

Figure 10 shows unweighted moving average filters in square shape of different sizes from 3x3 to 100x100. The filter size determines how much noise is removed but also how many details are lost. Filters with a larger area make the image more blurry and remove more details, but they also remove noise more evenly. The optimal filter sizes can vary a lot from image to image.

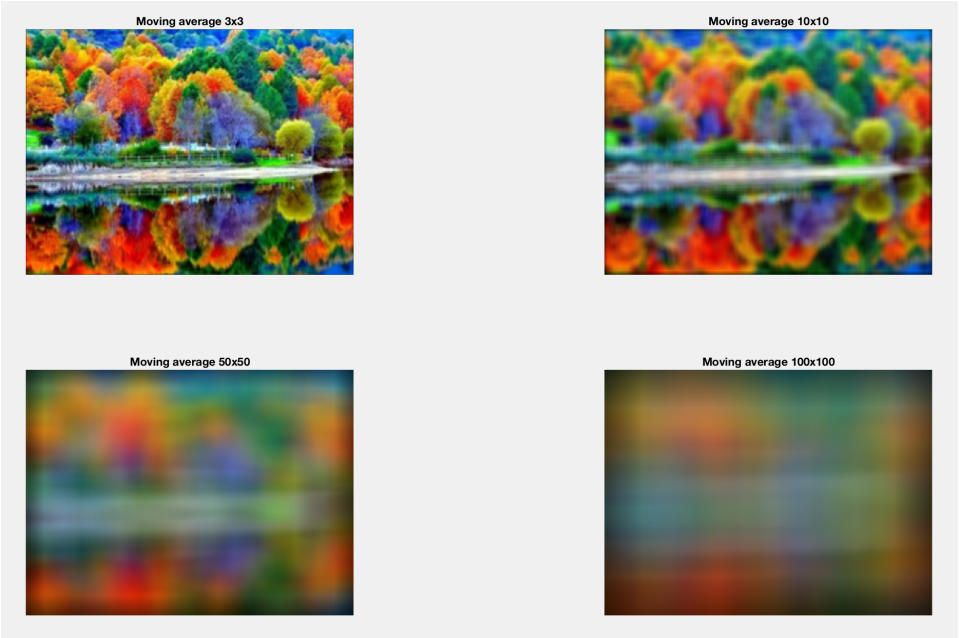

Figure 10: Square shaped unweighted moving averages in different sizes

Besides unweighted moving averages it is also possible to reward proximity to the target pixel by weighting close neighbors of the pixel more highly. Figure 11 shows a comparison of an unweighted horizontal kernel of 1 x 10 pixel size with a weighted horizontal kernel with weights [1, 2, 4, 8, 16, 32, 16, 8, 4, 2, 1]. The unweighted filtering gives the same importance to pixels that are much further away than close neighbors and results in a more blurry picture with less details.

In many cases it is a wanted effect to regard pixels in close proximity to the target pixel more highly than the ones that are further away. This can

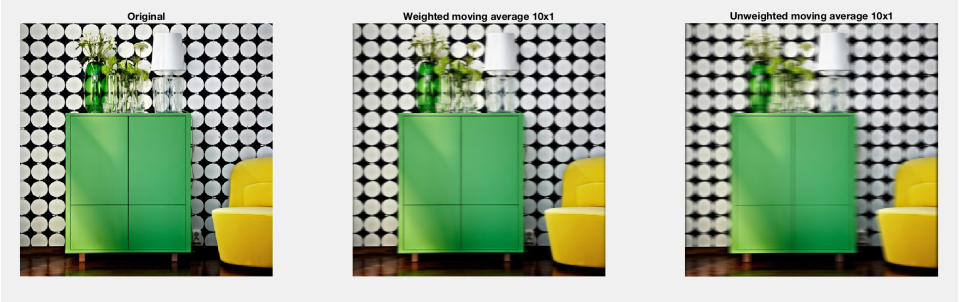

Figure 11: Weighted and unweighted horizontal moving average

elegantly be achieved by applying a gaussian kernel, like the ones depicted in figure 12. Each gaussian kernel has to parameters: kernel size and sigma. The kernel size determines the maximum area in which pixels are considered to compute the filtered value of the target pixel. The value of sigma determines, how much stronger close neighbors are weighted, than distant neighbors. A small sigma results in close neighbors being weighted much higher than with a large sigma.

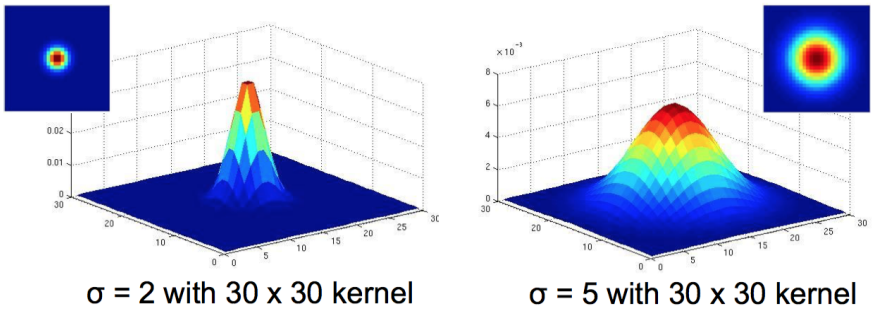

Figure 12: Two differently parameterized gaussian kernels

Figure 13 shows the effect of different kernel sizes of a gaussian filter. The left image used a kernel with area 10x10 pixels and the right image with a kernel of 30x30 pixels is clearly more blurry with more details and noise removed.

Figure 13: Gaussian kernels with different areas

As described before, the choice of sigma for gaussian kernels also influences their effect. This can be seen in figure 14.

Figure 14: Gaussian kernels with different sigmas

The inbuilt function *imfilter* in Matlab can use both custom filters like the weighted average and predefined filters like the gaussian on both RGB- and grayscale-images. Even though the kernel is 2-dimensional, the function can apply it sequentially on all three channels of the three-dimensional RGB-images.

It is important to normalize kernels in convolution filtering in order to keep values of the image generally in the same range of values like before. If normalization does not happen, the filtered image will have far higher values and eventually turn into a simple white image. This effect can be seen in figure 15.

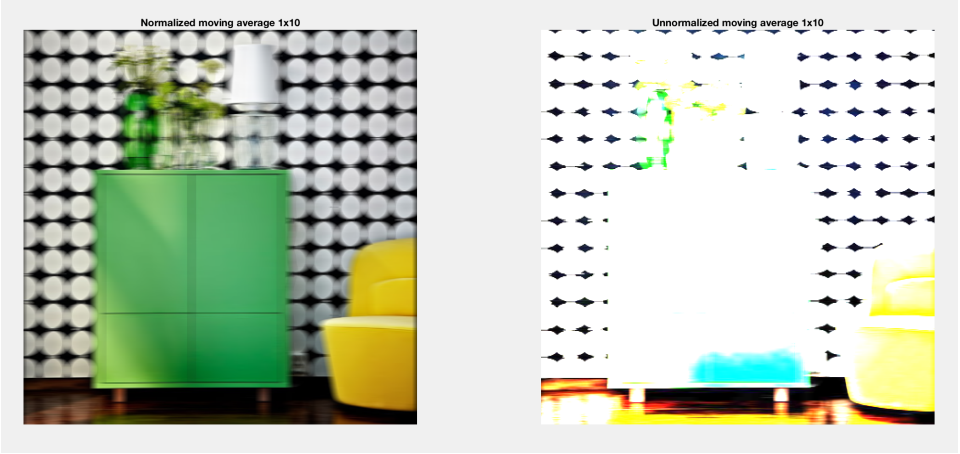

Figure 15: The importance of normalizing kernels

It is also possible to apply the same filter iteratively more than once to an image. The effect can be observed in figure 16

Figure 16: Applying filters multiple times

When calculating the absolute difference between a smoothed image and its original, an effect as in figure 17. Areas of the image that have very similar pixels stay don't change a lot when smoothing the picture and hence look black (near zero values) on the difference image. Only in areas of the image where pixels change more drastically, the difference shows higher values. This can be seen as some first step of edge detection.

Figure 17: Absolute difference between a smoothed image and its original

Exercise 4:

The wanted effect of swapping the two halves of the image can be achieved by simple concatenation with the Matlab function *horzcat*. For details see the file `exercise4.m`.

Figure 18: Image halves swapped

Exercise 5:

The inbuilt Matlab function *imbinarize* creates a logical array for an grayscale image I given a threshold t in which all pixels with values above t get mapped to a 1 and all other pixels to a 0.

Figure 19 shows binarizations for a grayscale image of a car for different thresholds. It can be seen that it depends heavily on the specific image which threshold results in a useful binarization. In this case, a threshold of 20 or 30 still maps most pixel to white pixels and does not yield a useful result. When moving the threshold closer to 1, most pixels get mapped to a black output, as the in the last image of figure 19.

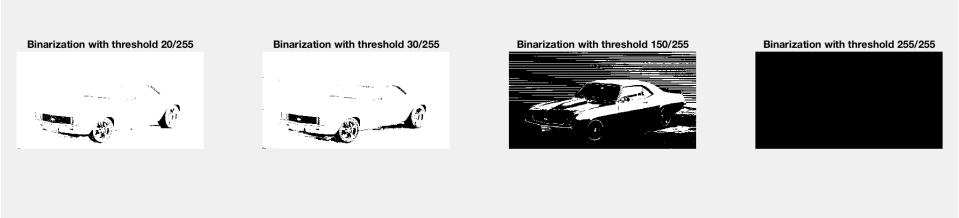

Figure 19: Image binarization with different thresholds

Figure 20 shows the pixel-wise product of the original image and its binarization with a threshold of 150/250.

Figure 21 shows the pixel-wise product of the original image and its inverted binarization with a threshold of 150/250.

Exercise 6:

In exercise 6, the pixels showing a hand are extracted from the black background of an image and copied onto a second image. The commands used for

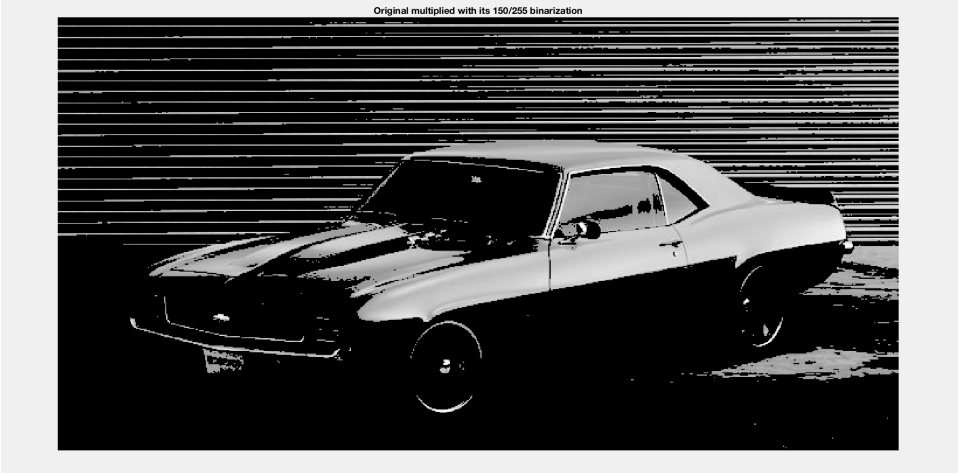

Figure 20: Original multiplied with its binarization

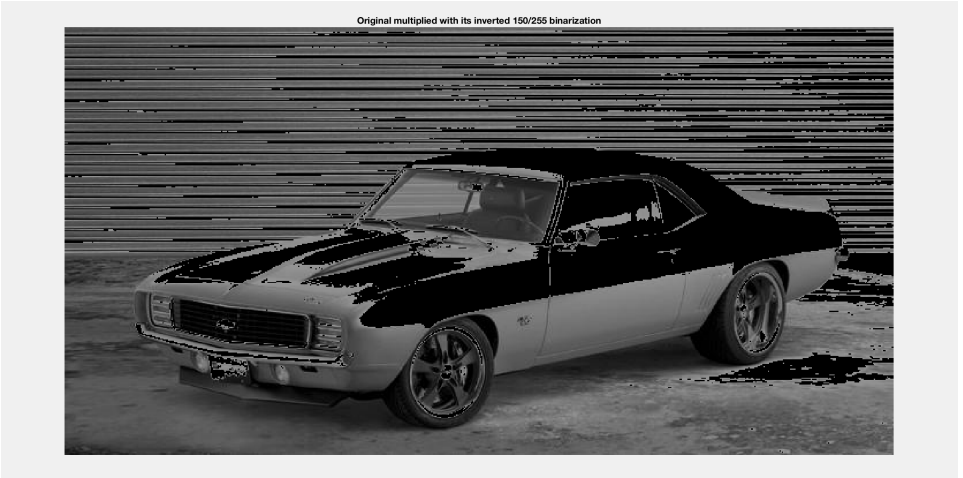

Figure 21: Original multiplied with its inverted binarization

this process can be found in `exercise6.m`. The resulting image is depicted in figure 22.

Figure 22: Hand copied in front of tower