Universitat Politècnica de Catalunya
Universitat de Barcelona
Universitat Rovira i Virgili

Master in Artificial Intelligence

Computing vision

Image smoothing and simple geometric operations in Matlab

Authors:
Johannes Heidecke
Alejandro Suárez Hernández

October 2016

Contents

List of Figures

1 Creating image of 3 channels

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

2 Displaying color images

The image *chairs.jpg* consists of the three different color channels of the RGB spectrum: red, green and blue. Pixels 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 gray scale 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 an 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 it can be seen in figure 2.

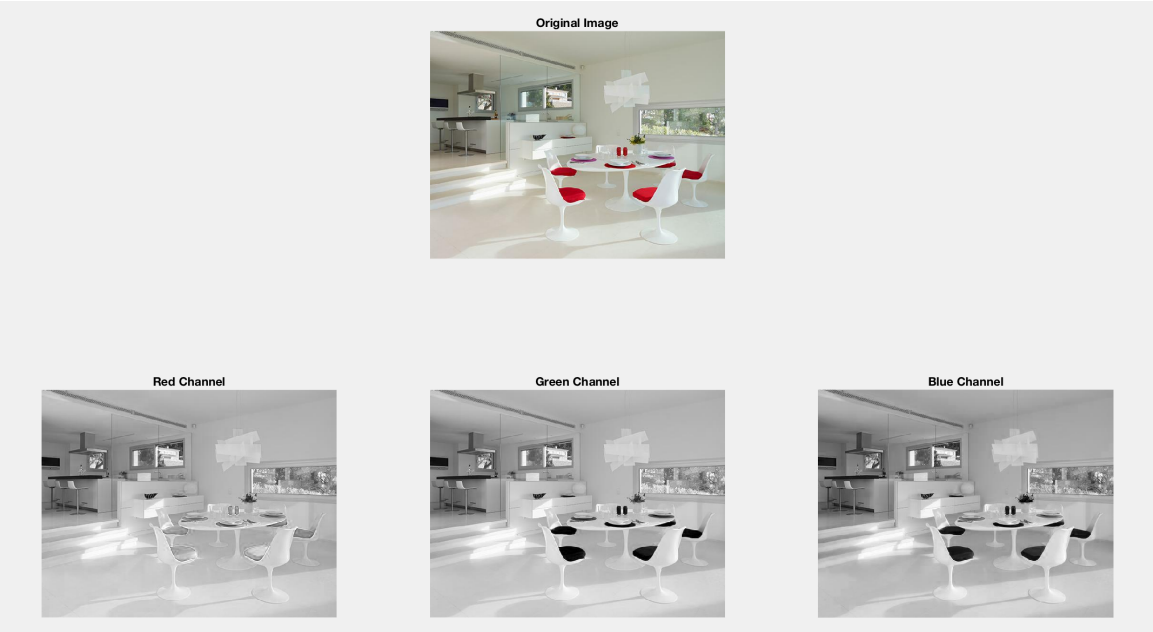

Figure 2: Original Image and its three RGB channels

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.

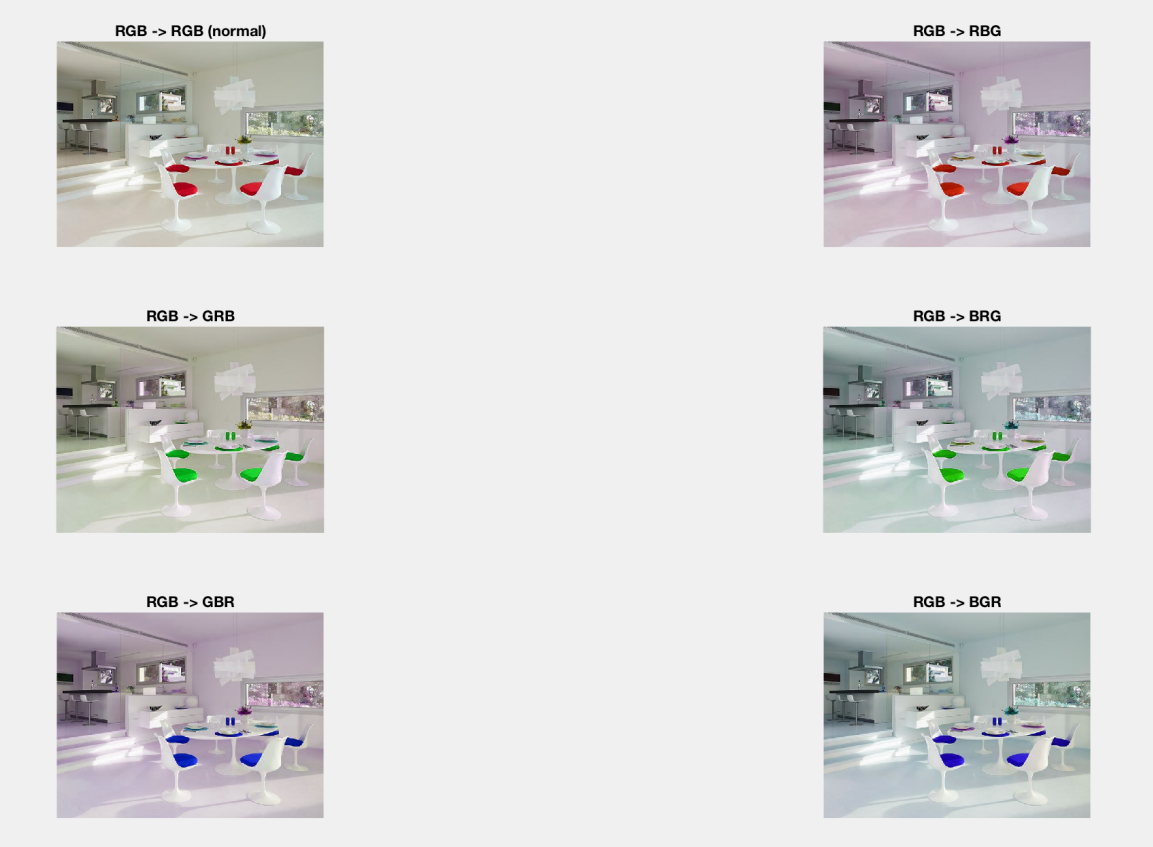

Figure 3: Interchanging the RGB channels

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

3 Managing different size and filters

Scaling down an image reduces not only the size but also the details of the image. This can be seen in figure 5. While with scale factor of 0.5 we can still appreciate most of the details of this particular image, after rescaling it to 0.01 the original size almost all details are lost.

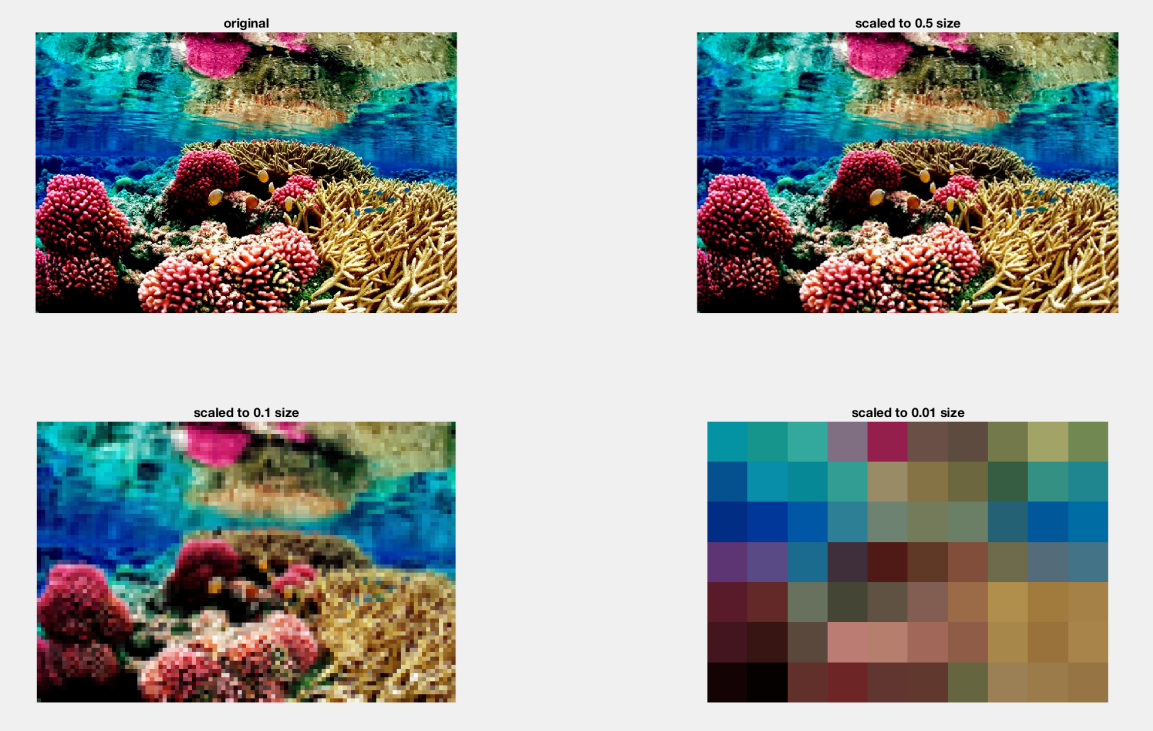

Figure 5: Resizing the original image to smaller sizes

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 one of the effects of combining the values of several adjacent bins into one in order to allow a smaller size.

It is possible to return the smaller image back to its original size. As figure 7 shows, the restored image keeps lacking the same amount of details of the original image. The details that were lost in the small image cannot be retrieved later. The only improvement is, that the restored image looks smoother and less pixelated, i.e. pixels boundaries are not perceptible with the same zoom factor as before. This is thanks to the bicubic interpolation method followed to scale the image back to its original size.

This effect can be observed better when we zoom in a smaller area of the image, as in figure 8. This figure shows, that when scaling down to 0.1 of original size, most details are lost and the image looks very pixelated. When restoring the 0.1 image back to its original size with a bicubic interpolation, the pixelated look disappears, but the former details can not be recovered.

As an alternative for removing image details different smoothing filters can be applied. One option is to apply evenly weighted moving averages that simply add all values in the

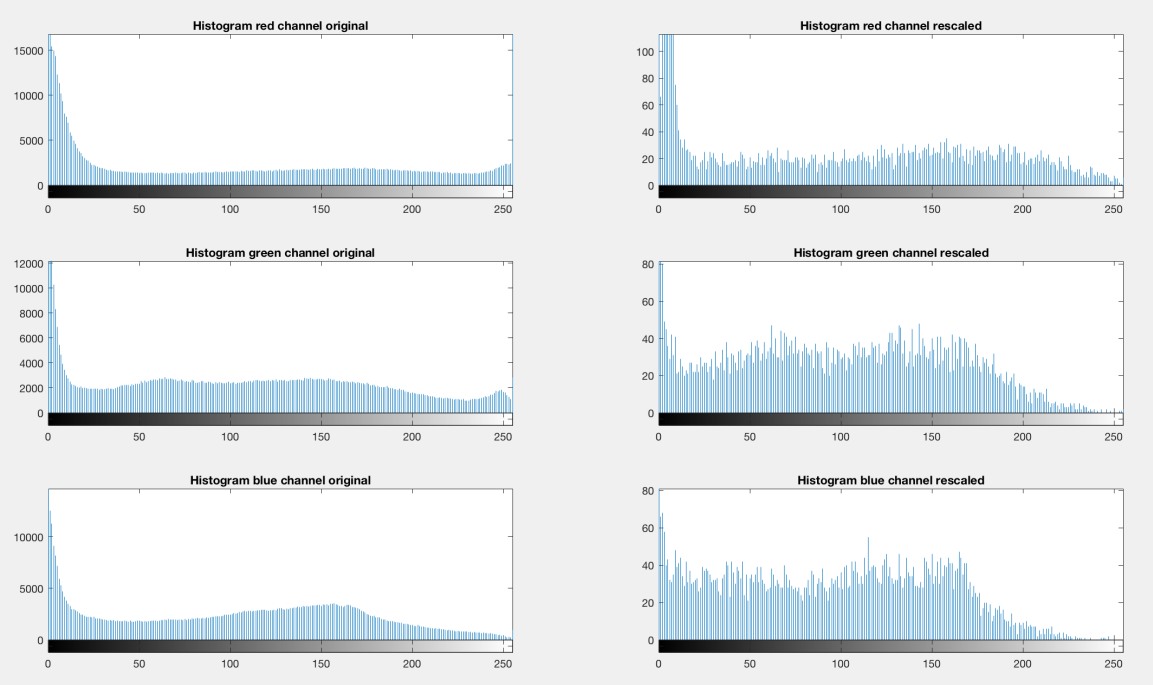

Figure 6: Change of the RGB histograms after resizing to 0.1 original size

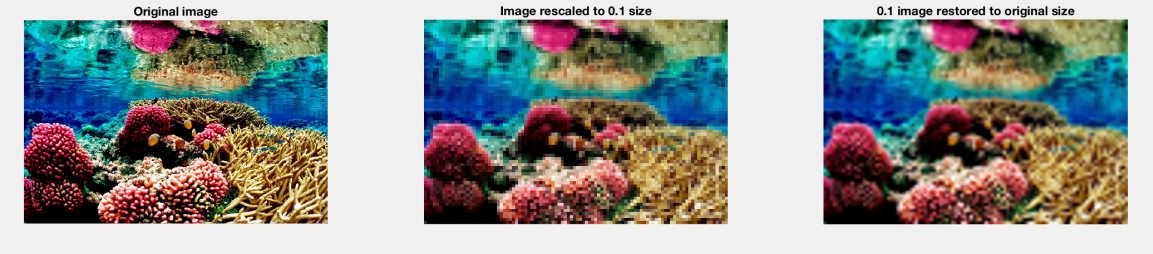

Figure 7: Resizing to 0.1 size and then restoring to original size

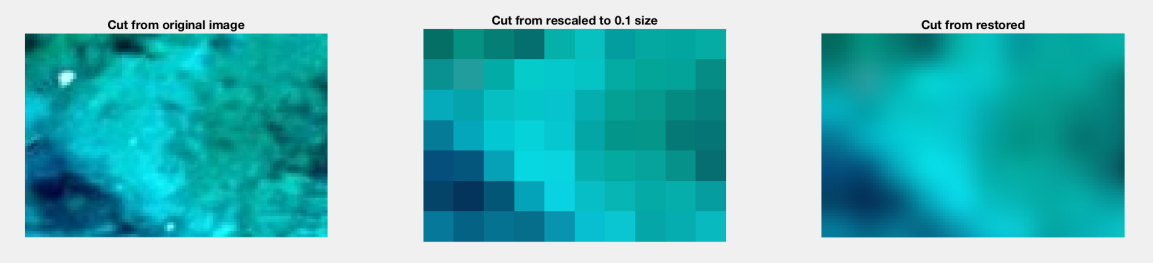

Figure 8: Resizing to 0.1 size and then restoring to original size (Zoomed in)

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 1x50 pixel and a vertical filter of size 50x1 is applied to all pixels in the image. The former results in an effect that looks like the camera had been moving horizontally while taking the picture, while in the latter the movement is horizontal (this is known as motion blur in most photography/art suites like GIMP or Photoshop). While both filters will remove some kind of noise in the image or add some artistic effect, their results still look very different.

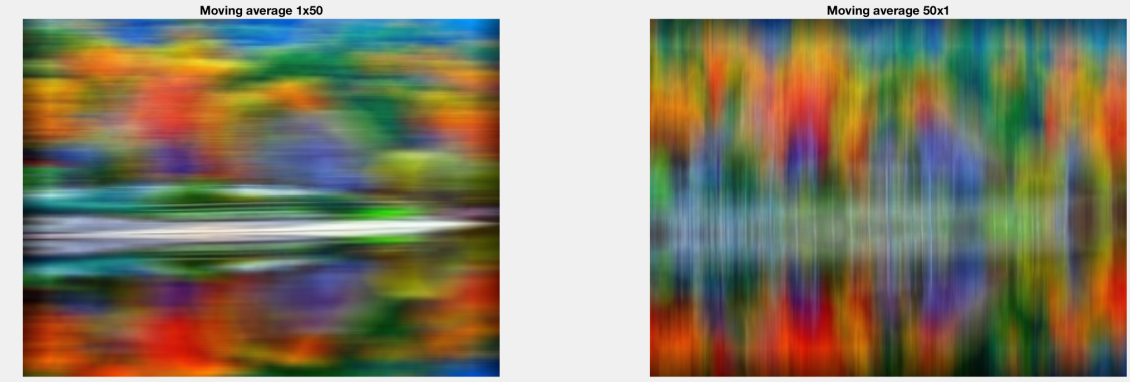

Figure 9: Horizontal and vertical evenly weighted moving average

Figure 10 shows evenly weighted 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 have a greater denoising power. The optimal filter sizes can vary a lot from image to image.

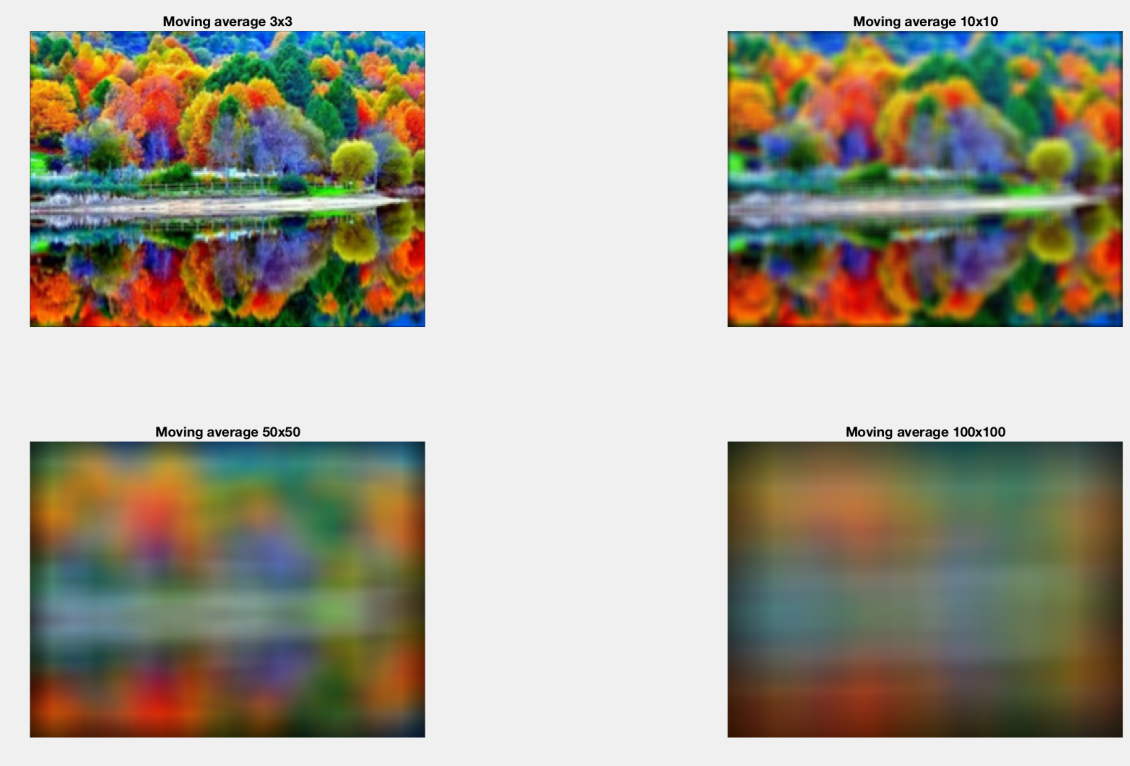

Figure 10: Square shaped evenly weighted moving averages in different sizes

Besides evenly weighted 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 the comparison of an evenly weighted horizontal kernel of 1 x 10 pixel size against a weighted horizontal kernel with weights [1, 2, 4, 8, 16, 32, 16, 8, 4, 2, 1]. The evenly weighted 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.

Figure 11: Weighted and evenly weighted horizontal moving average

In several cases it is desired to weight neighbors closer to the target pixel higher than the ones that are further away. This can be achieved elegantly by applying a Gaussian kernel, like the ones depicted in figure 12. If the Gaussian filter is isotropic (i.e. same behavior in every direction), it has two parameters: kernel size and sigma. Otherwise, we would need to consider two additional parameters. 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. It is very common to choose the kernel size as a function of sigma (e.g. three or five times sigma) to avoid having a prematurely truncated Gaussian kernel.

As a side note it is worth noticing that one of the main advantages of the ideal non-truncated Gaussian filter is the absence of side lobes in the frequency response. This is, it acts as a reasonably selective low-pass without oscillating behavior for high-frequency bands.

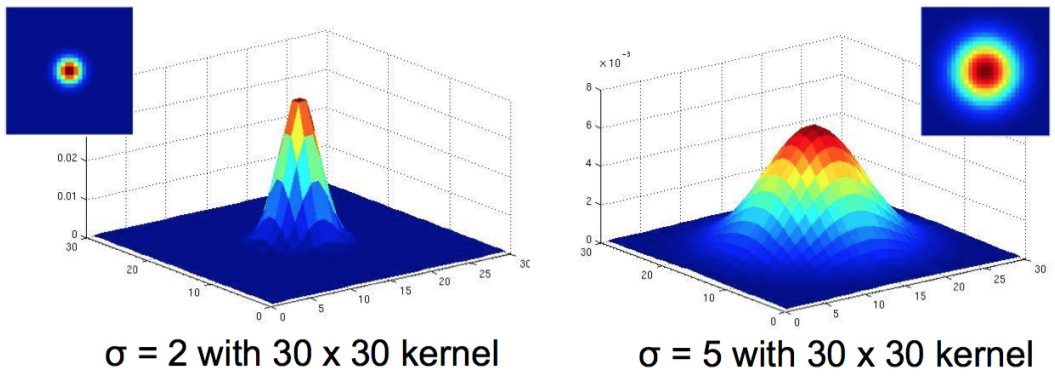

Figure 12: Two differently parametrized Gaussian kernels

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. This is because the 10x10 kernel is a prematurely truncated sigma.

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

Figure 13: Gaussian kernels with different areas

Figure 14: Gaussian kernels with different sigmas

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

Usually it is desired to normalize kernels in convolution filtering in order to keep the values of the resulting image bounded. If the filter is not normalized, the output is very likely to saturate. This effect can be seen in figure 15.

It is also possible to apply the same filter iteratively more than once to an image. The effect can be observed in figure 16. It is worth noticing that convolving more than once an image with the same kernel will be similar to applying an Gaussian filter (anisotropic in the general case) from the beginning. This is because of the central limit theorem (we can think of convolving the same filter with itself several time as summing several identically distributed random variables).

When calculating the absolute difference between a smoothed image and its original, we get an effect similar to that in figure 17. Areas of the image that have very similar

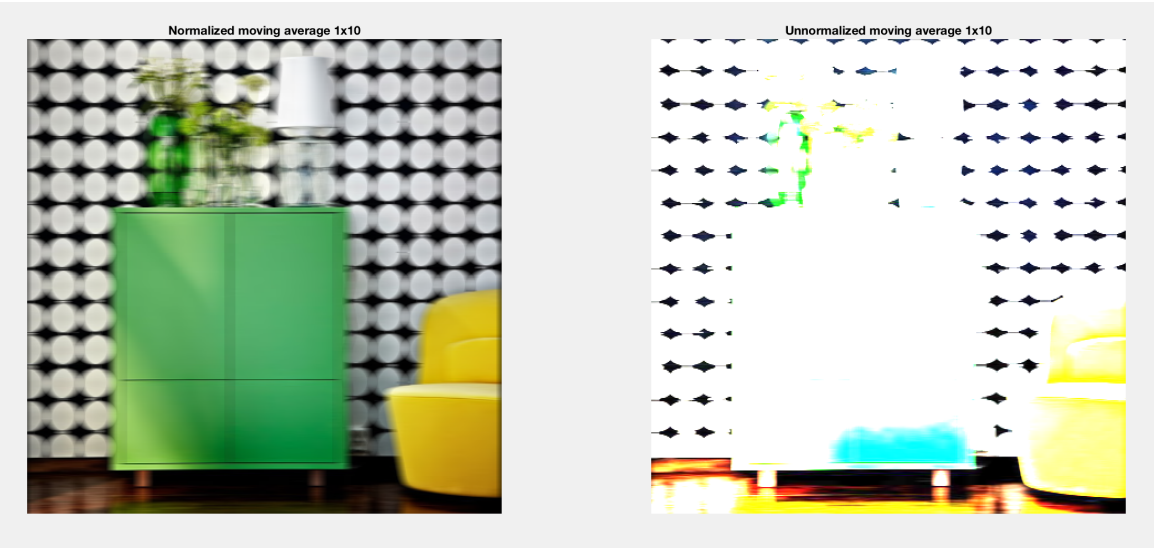

Figure 15: The importance of normalizing kernels

Figure 16: Applying filters multiple times

pixel values in both images look darker (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 an idea for an edge detection technique. In fact, DoG (Difference of Gaussians) is a well known algorithm for edge detection which puts into practice a similar concept.

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

4 Simple geometric operations

The desired 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

5 Image binarization

The built-in 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 gray scale image of a car for different thresholds. It can be seen that the optimum threshold depends heavily on the specific image. In this case, a threshold of 20 or 30 still maps most pixel to white values and does not yield a useful result. When moving the threshold closer to 1, most pixels get mapped to a white output, as the in the first image of figure 19. Conversely, for very high thresholds (like the extreme 255) maps almost every pixel to black.

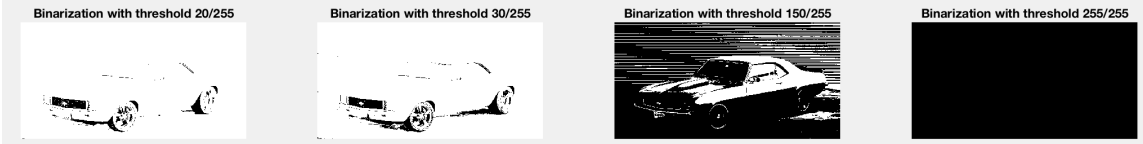

Figure 19: Image binarization with different thresholds

Figure 20 shows the pixel-wise product of the original image and its binarization with a threshold of 150/255.

Figure 21 shows the pixel-wise product of the original image and its inverted binarization with a threshold of 150/255.

6 Treating color images

In exercise 6, the pixels showing the hand are separated from the black background of the original image and copied onto a second image. The commands used for this process can be found in `exercise6.m`. The resulting image is depicted in figure 22.

Figure 20: Original multiplied with its binarization

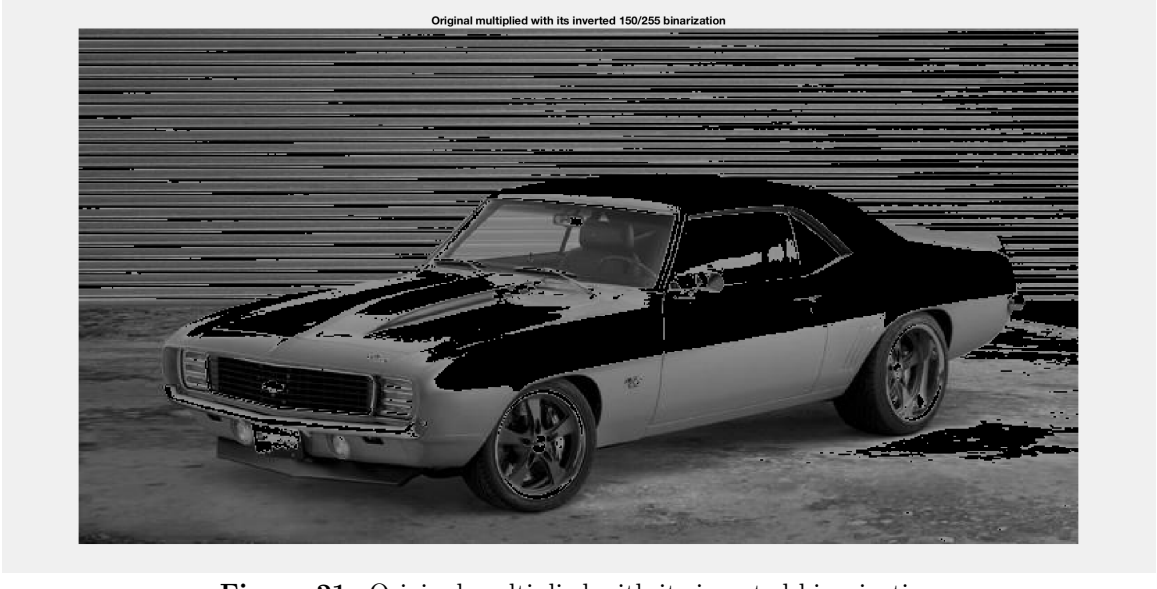

Figure 21: Original multiplied with its inverted binarization

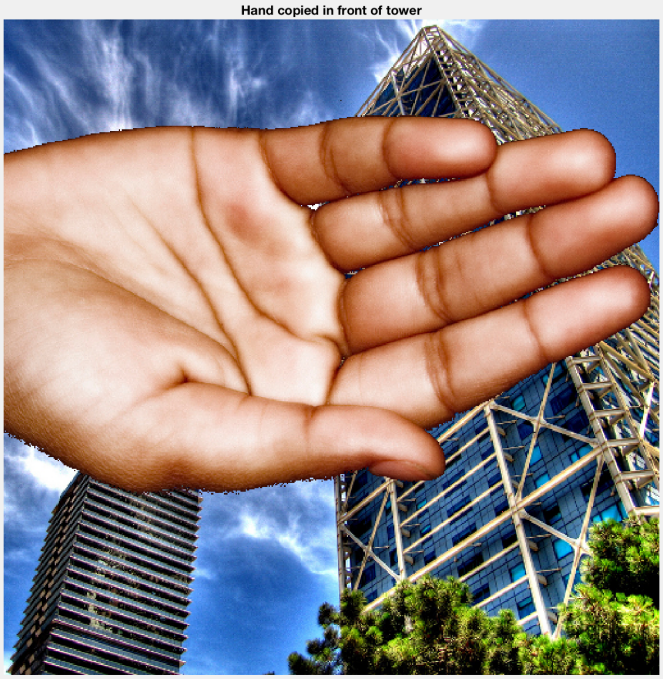

Figure 22: Hand copied in front of tower

Appendices

A Delivered code files

Here we list all the delivered script and function files delivered for this practice. We include relevant observations. For full insight, we refer the reader to the source code.

- `exercise1.m`
- `exercise2.m`
- `exercise3.m`
- `exercise4.m`
- `exercise5.m`
- `exercise6.m`