Homework1: Fast Local Filtering

# Histogram matching

## Algorithm

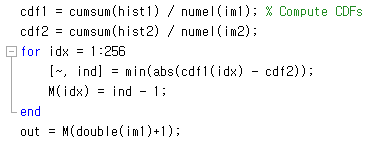
Histogram matching is concerned with transforming one image’s histogram so that it looks like another. The basic principle is to compute the histogram of each image individually, then compute their discrete cumulative distribution functions (CDFs). Once we calculate the CDFs for each image, we need to compute a mapping that transforms one intensity from the first image so that it is in agreement with the intensity distribution of the second image.

CDF1(I1) = CDF2(I2)

There may be a case where we won’t get exactly an equality, so we need to find the smallest absolute difference between CDF1(I1) and CDF2(I2). In other words, for a mapping M for each entry of I1, we must find an intensity I2 such that:

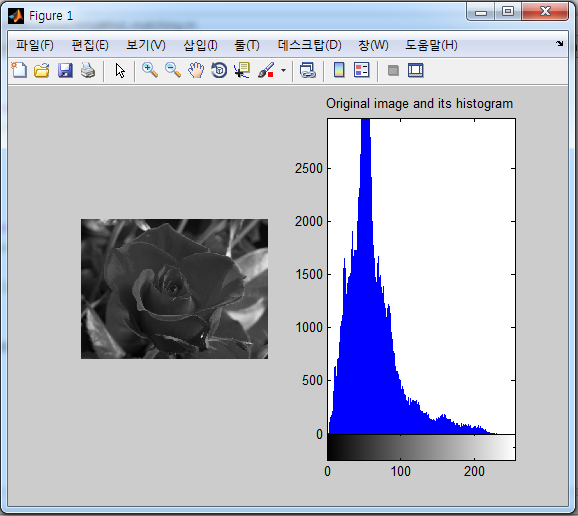
M(I1) = argminI2ϵ[0,255] |CDF1(I1) – CDF2(I2)| ꓯI1 ϵ [0,255]

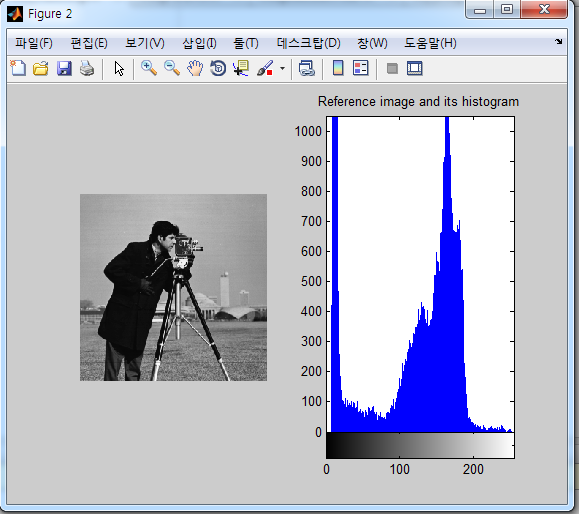
We would do this for all 256 values, and we would produce mapping. Once we find this mapping, we have to apply this mapping on the first image to get it to look like the intensity distribution of the second image like this:

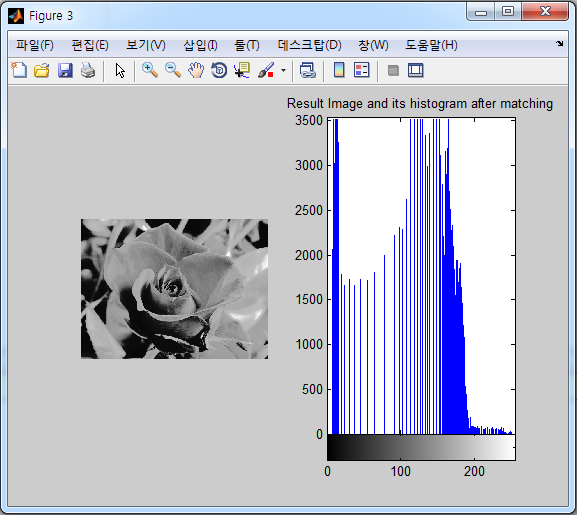


“out” contain our matched image where it transforms the intensity distribution of the first image to match that of the second image. The intensity range of im1 spans between [0,255], but Matlab’s indexing for arrays starts at 1. Therefore, we need to add 1 to every value of im1 so we can properly index into M to produce our output. However, im1 is of type uint8, and Matlab saturates values should we try and go beyond 255. As such, to ensure that we get to 256, we must cast to a data type that is beyond 8-bit precision so we use ‘double’, then when we add 1 to every value in im1, we will span between 1 to 256 so we can properly index into M. Also take not that when we find the location that minimizes the difference, we also must subtract by 1 as the data type sans form [0, 255].

## Results







# 2D Integral Image

Perform O(1) time box filtering with 2D integral image. The total runtime should be unchanged when the window size varies.

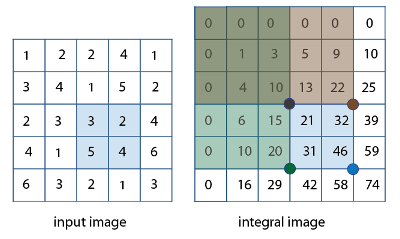
## Method 1:

Compute integral image

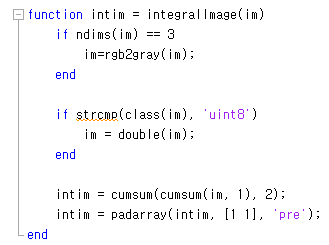
Compute I0(x,y) = S(x+a, y+b) – S(x-a-1, y+b) –S(x+a, y-b-1) + S(x-a-1, y-b-1)

Compute average of image I0(x, y) = I0(x,y) / N where N = (2a+1)\*(2b+1)

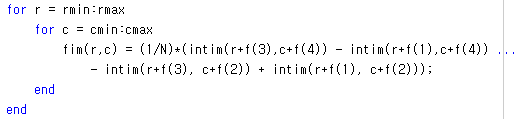
Using an integral image, we can rapidly calculate summations over image subregions. Integral image facilitate summation of pixels and can be performed in constant time, regardless of the neighbourhood size. The following figure illustrates the summation of a subregion of an image.



The integral image is calculated as follow:

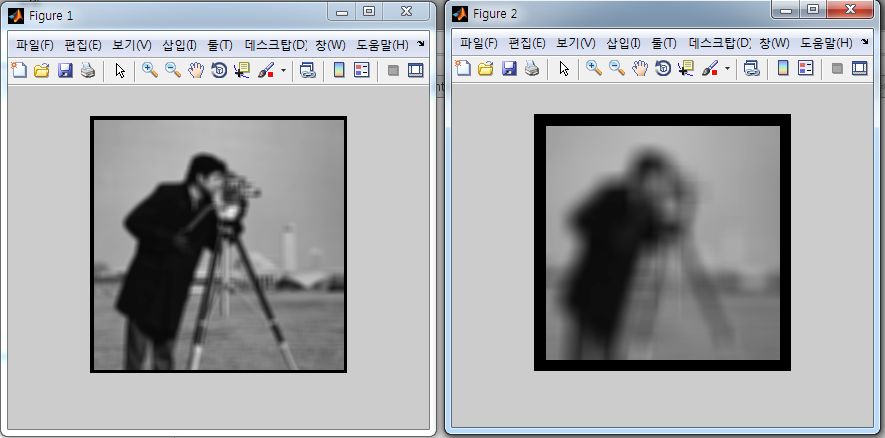


After getting integral image, we can filter the it:



## Method 1 results:

Using 2 different filter size, the first one is f1 = [-3 -3 3 3] (where N1=7x7) and the second one is f2 = [-11 -11 11 11] (where N2=23x23).



Execution time is similar in both cases:



## Method 2:

Compute integral image along horizontal direction S1(x, y) =



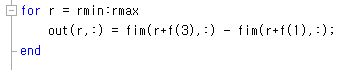
Compute I1 = S1(x, y+b) – S1(x, y-b-1)



Compute integral image along vertical direction S2(x, y) =



Compute Io(x, y) = S2(x+a, y) – S2(x-a-1, y)

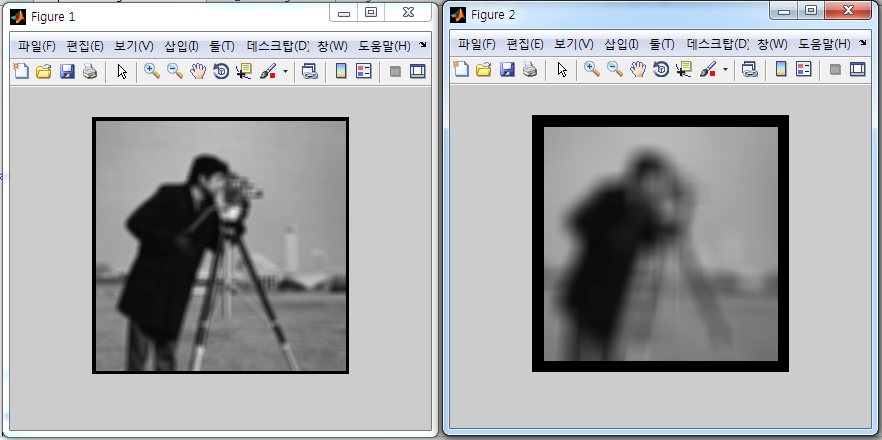


Compute filtered image Io(x, y) = Io(x, y) / N where N = (2a+1)\*(2b+1)



## Method 2 results:

Using 2 different filter size, the first one is f1 = [-3 -3 3 3] (where N1=7x7) and the second one is f2 = [-11 -11 11 11] (where N2=23x23).



Execution time is similar in both cases:

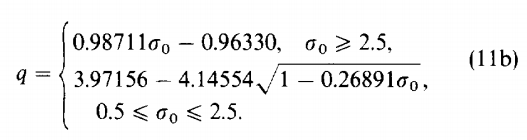


According to results, we found that the second method runs a bit faster than the first one.

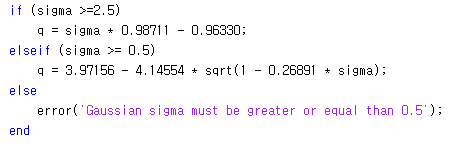
# 2D Gaussian filtering

## Method 1: Perform O(1) time 2D Gaussian filtering using the recursive implementation.

Calculation of parameter q bases on the paper that

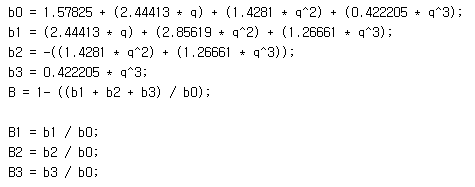


Matlab code:



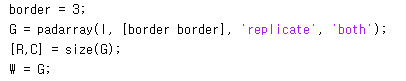
Calculate the coefficients of the Gaussian filter

The parameter q and the coefficients were determined with the Z transform:



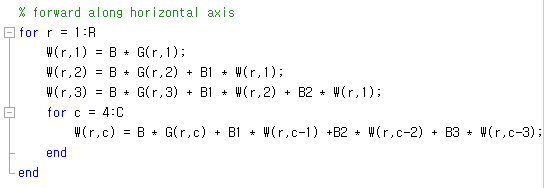
Edge

Each convolution poses the problem of management of the edges. For each pixel, the calculation of the filter requires the presence of at least 3 pixels preceding and 3 successive pixels. This implies that the first 3 columns (or rows) cannot be calculated. To resolve this problem, the image is enlarged by a quantity of pixels sufficient to apply the algorithm, or 3 pixels.



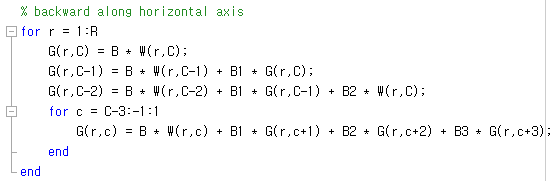
Apply forward equation on columns

An image signal is a 2D dimensions. Using the properties of the Gaussian to be separable, the algorithm is applied along a direction and the outcome is re-applied along the other direction. Each row of the image is considered as a single vector whose elements are the pixels of the columns for the data row. Therefore, the rows are constant while the columns are variable:



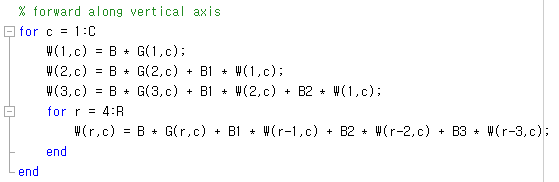
Apply backward equation on the columns:

Equation is applied backwards form the last element to the first:

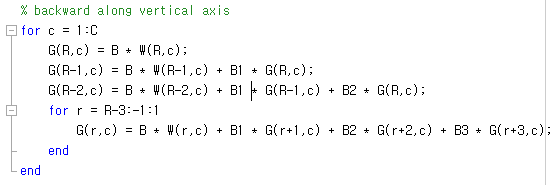


Apply forward equation on rows

This time, the equation uses the rows as variable and the columns as constant.



Apply backward equation on rows

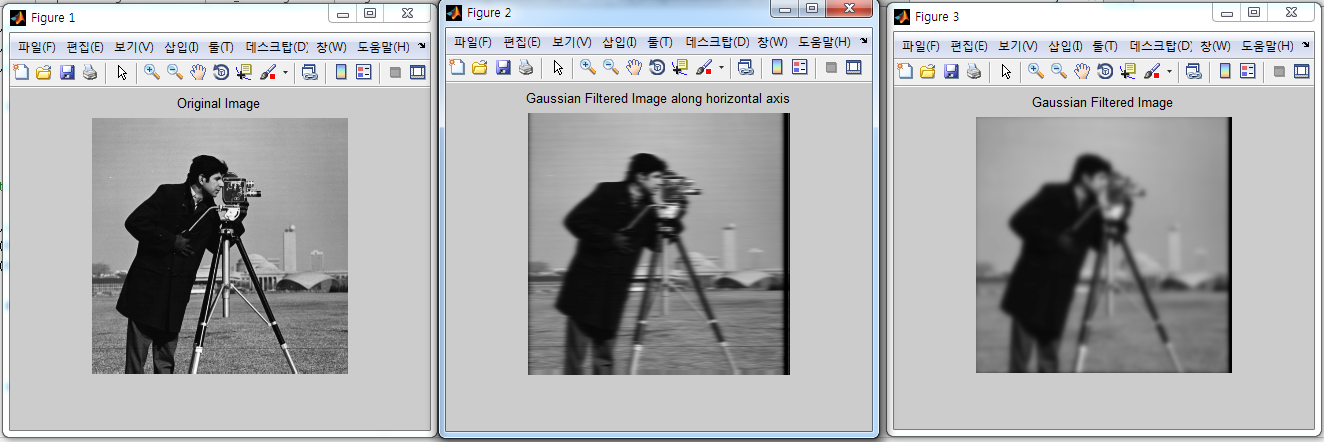


Restore the image size

Now, the image is filtered along both directions. The previously added edges need to be removed to get the final result filtered image.

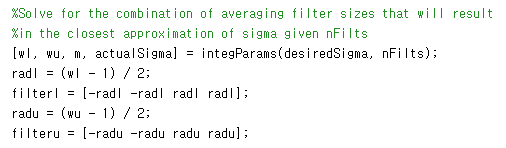


Results



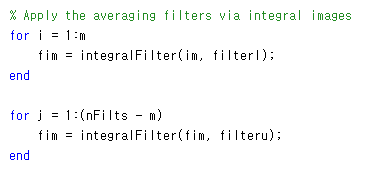
## Method 2: Perform 2D Gaussian filtering by repeatedly applying 2D integral image.

First, we need to solve the combination of averaging filter sizes that will result in the closest approximation of sigma given by number of filter



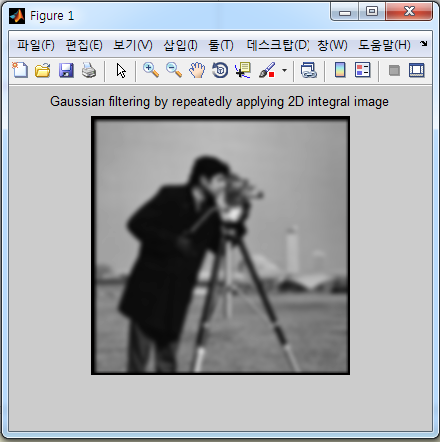
Wl is the width of smaller averaging filter to use, wu is the width of larger averaging filter to use, these widths are always odd, m is the number of filterings to be done with the smaller averaging filter. The number of filterings to be done with the larger filter is (nFilts – m). And actualSigma is the actual standard deviation of the approximated Gaussian that is achieved.

Then we filter the image by averaging multiple filter using integral image.



As the above code, we need ‘m’ iterations to filter with smaller averaging filters and ‘(nFilts – m) ‘ iterations with larger averaging ones, so in total we need m + nFilts – m = nFilts iterations or the number of iterations is equal to the number of filters.

Results:



Comparing results between 2 methods using same sigma = 2.6

|  |  |
| --- | --- |
|  |  |