Project Report “Exposure Fusion for HDR Effects”

Patrick Renelli

The "Exposure Fusion for HDR Effects" project asked to implement a version of the algorithm presented in the article "Exposure Fusion by Tom Mertens Jan Kautz Frank Van Reeth" which describes one of the possible HDR implementations based on weight maps generated by various input images, used to weight its pixels, based on contrast, saturation and exposure, will finally overlay and merge the supplied images into one final image.

Implementation

For the realization of the project it was decided to create the "HDRElaborator" class in python which implements all the functions necessary for processing, using the cv2 and numpy libraries.
The implemented functions are:

- loadImage: used to load and resize the images taken as input,
- align: used to train loaded images,
- lowPassFilter: used to resize and apply a Gaussian filter to the images,
- gaussPyramid: used to calculate Gaussian pyramids,
- laplPyramid: used to calculate the Laplacian pyramids,
- findImportance: used to calculate image weights based on contrast, saturation and exposure,
- overlap: used to calculate image fusion based on weighted maps,
- getHDRImage: used to call the above functions and obtain the processed image

Initially by calling the constructor of the HDRElaborator class, to which we pass an array containing the names of the input images and the depth of the pyramids that we will subsequently create, after having created an object of the class just mentioned, we will call its getHDRImage method which will call the various methods required for creating the HDR image.
The loadImage function will be called, which after having loaded the various files of the images taken as input will check that at least 2 files have been loaded, with only one image the execution of the algorithm will be useless, otherwise the function will return, subsequently yes calculates the average size of the photos, checks whether this calculated size is compatible with the future resizes that will need to be done when calculating the various Gaussian and Laplacian pyramids and the final fusion to obtain the final image. Compatibility means that the images must have dimensions whose result of the modulus for the 2nd depth imposed on the algorithm is equal to 0, therefore we will find this new one dimension compatible with our needs and we perform a resize of all the passed images, going to use a different interpolation in case we are going to reduce or increase the dimension, in the first case we will use an interpolation, always provided by cv2, called INTER_CUBIC, while in the second case we will use the INTER_AREA. Therefore, if we have loaded at least 2 files, we proceed to align the images, using the process function of the AlignMTB class which performs a "soft alignment" managing to cancel the smallest phase shifts. Once the photos are aligned, by calling the overlap function we will execute the actual algorithm. We start by invoking the findImportance function which, after having built a "singleImportance" matrix of the size of the photos totally initialized to 0,

To calculate the contrast, saturation and exposure values as required in the paper, proceed as follows.
For the**contrast**we convert the analyzed photo in grayscale through the cv2.cvtColor function, passing it as required conversion "COLOR_BGR2GRAY", we take the absolute value of the image to which we apply a Laplacian filter, with a final depth set to "CV_32F". The Laplacian filter is a filter that calculates the second derivatives of an image, thus measuring how quickly the intensities change, it is usually used to identify the edges and textures present in the image, finally we add the matrix to the "weight" matrix related to contrast values.
We want to select only the most saturated colors, since excessive exposure leads to desaturated colors, which we don't want to include in our final image. To calculate the relative weight of the**saturation**we will calculate the derivative of the various pixels and add the value of the weighted saturation matrix to the "weight" matrix. In calculating an optimal exposure we want to keep all those values that in a range between 0 (which represents an underexposed image) and 1 (which represents an overexposed image), are around 0.5. To calculate the weight
of the**exposure**of the individual pixels we will take advantage of the function indicated in the paper, which we translate into python as "exp(-(([image matrix] - 0.5)**2)/(2*(0.2**2)))", going to calculate the product of the matrices obtained to have a single matrix that can be added to "weight". When we have calculated its weights for each image, we divide the weighted maps with the summed importance of the various maps and normalize by multiplying by 255.

Let us now compute the Laplanic and Gaussian pyramids as required.
For the first we start by calculating the Gaussian pyramids of the individual images, after which at each level of these Gaussian pyramids we apply the lowPassFilter function, passing it False as an exp parameter.
Before talking about the implementation of this function we define a Gaussian kernel as a 2d matrix and it is generally used to make a low pass filter where the values are derived from the "cv2.getGaussianKernel" function which allows us to obtain the
coefficients of the Gaussian filter, in our case we will use an aperture of 5 and a Gaussian standard deviation calculated by default.
The lowPassFilter function accepts as input the image to be processed and a boolean exp which determines its operation:

- if exp = True we will apply the "cv2.filter2D" function which will apply the Gaussian filter to the input image and we will reduce the image size through "cv2.resize" to a

  quarter of its original size since we use the coefficients fx and fy of value 0.5, thus halving both the dimension of the x-axis and of the y-axis,

- if exp = False we will apply the "cv2.filter2D" function as above and then the "cv2.resize" function this time with coefficients fx and fy of value 2, in order to bring the image back to the original size

After that we compute the difference between the last scaled image of the Gaussian pyramid and the newly processed image and add it to the list we will return to.
To calculate the Gaussian pyramids let's apply the lowPassFilter function, this time with exp True in order to apply the Gaussian filter and reduce the output size. At this point, for each level we will build

the "moreDetails" matrix initialized to 0, and for each image we will first normalize the Gaussian pyramids, bringing all their values to the range [0,1], stack the Gaussians and we multiply the Gaussians with the laplanic images, at this point we add the result to the previous elaborations, in this way with each iteration I will go to capture more and more details, saving everything in the "moreDetails" matrix. Finally we add the images generated by each iteration of the inner for loop for the individual layers across all images to the pyr array. The code I elaborated was developed starting from several articles and implementations already present online, collecting from each one as much information as possible on the use of useful functions present in the two libraries used in the implementation and examples of implementations and final results useful for making sure of the goodness of the images processed by my algorithm

Examples of execution

The 3 starting images used as an example have a size of 1200x800

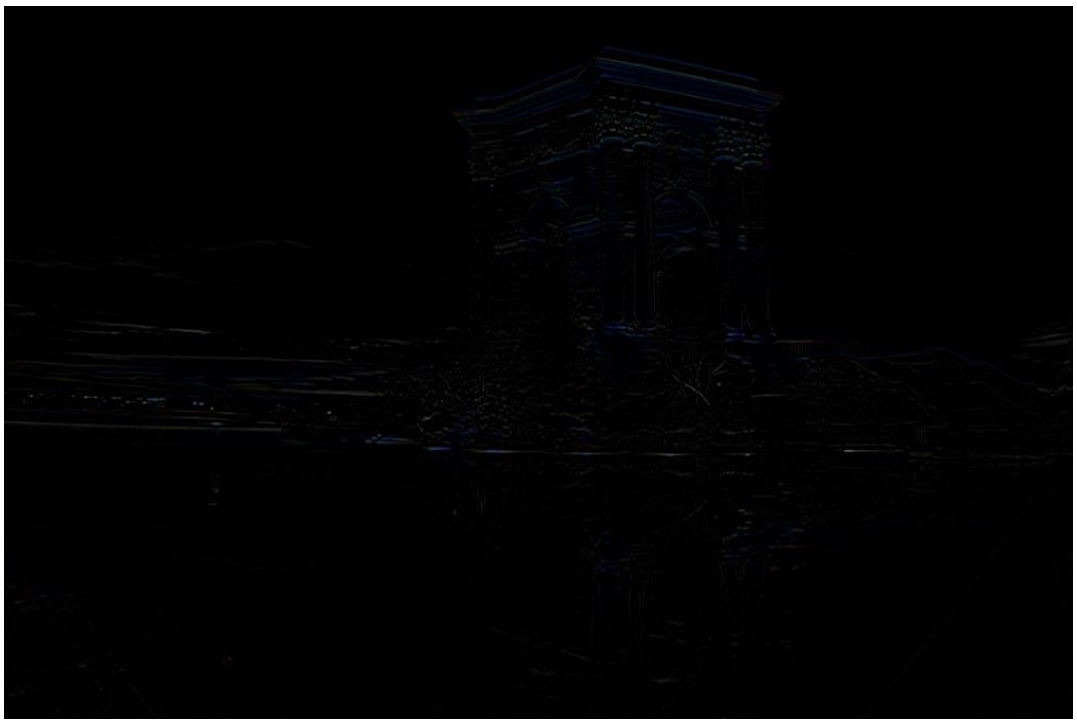
(1) Image overexposed

(2) Image with intermediate exposure

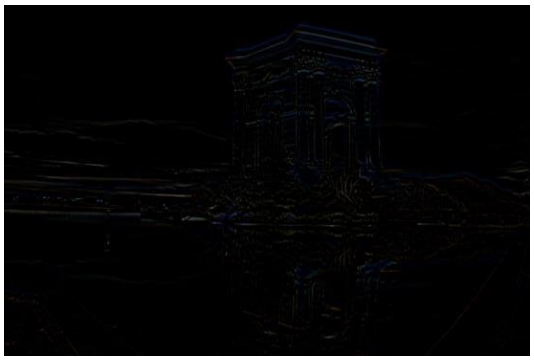
(3) Image underexposed

Laplanic pyramids

1) We calculate the laplanic pyramids of the different photos using 4 as the number of levels



The first image that will form the Laplacian pyramid will have size 1200x800



The second image that will form the Laplacian pyramid will have a size of 600x400

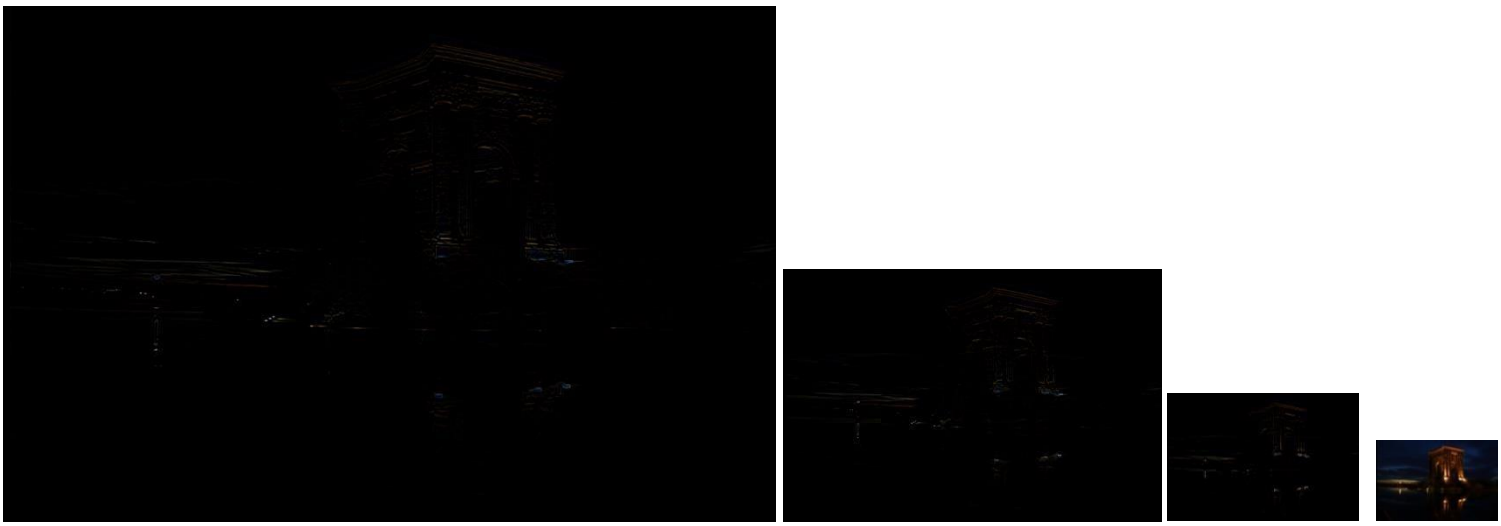
The third image it will form the Laplacian pyramid will have 300x200

The fourth image that will form the Laplacian pyramid it will have 150x100

2)

1200x800 600x400 300x200 150x100

(3)



1200x800 600x400 300x200 150x100

After calculating the laplanic pyramid for each photo, let's calculate the various Gaussian pyramids.

Gaussian pyramids

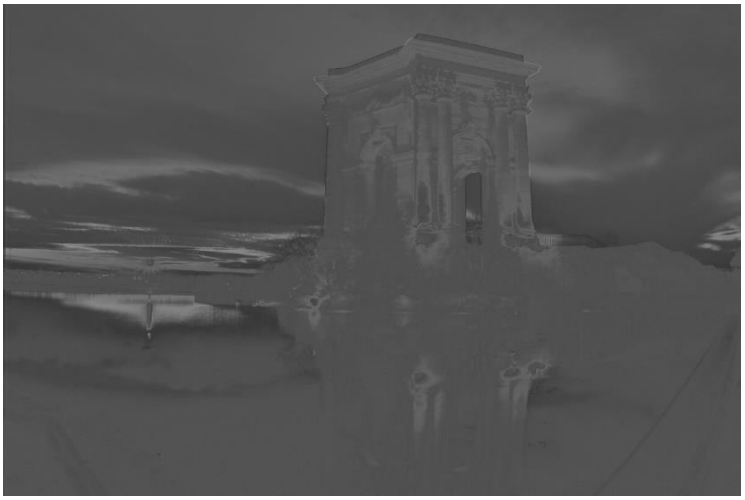
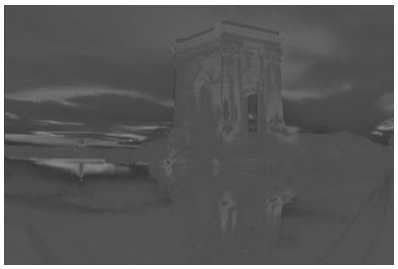
We calculate the Gaussian pyramids of the different photos using 4 as the number of levels

(1)



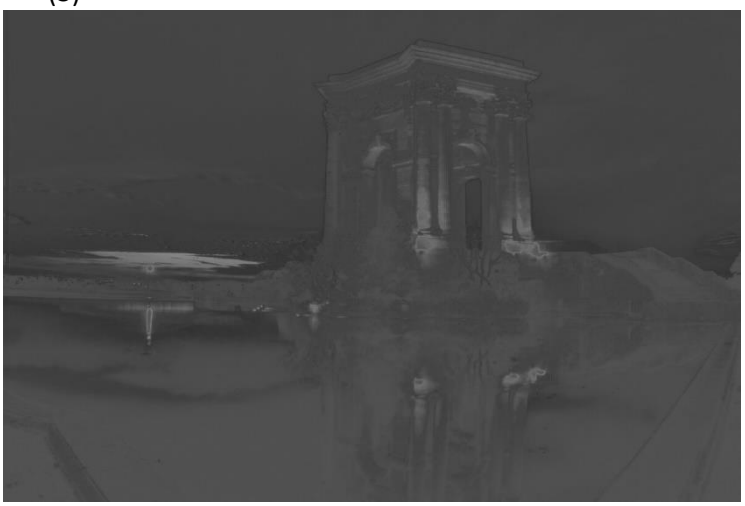

1200x800 600x400 300x200 150x100

(2)

1200x800 600x400 300x200 150x100

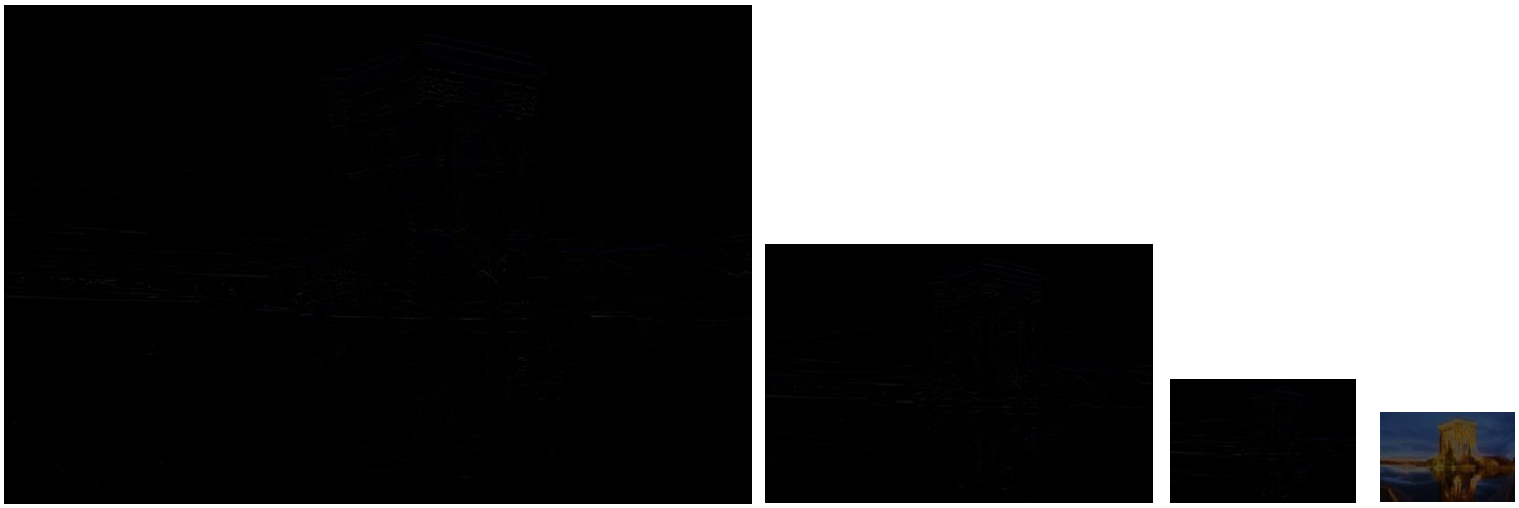
(3)




1200x800 600x400 300x200 150x100

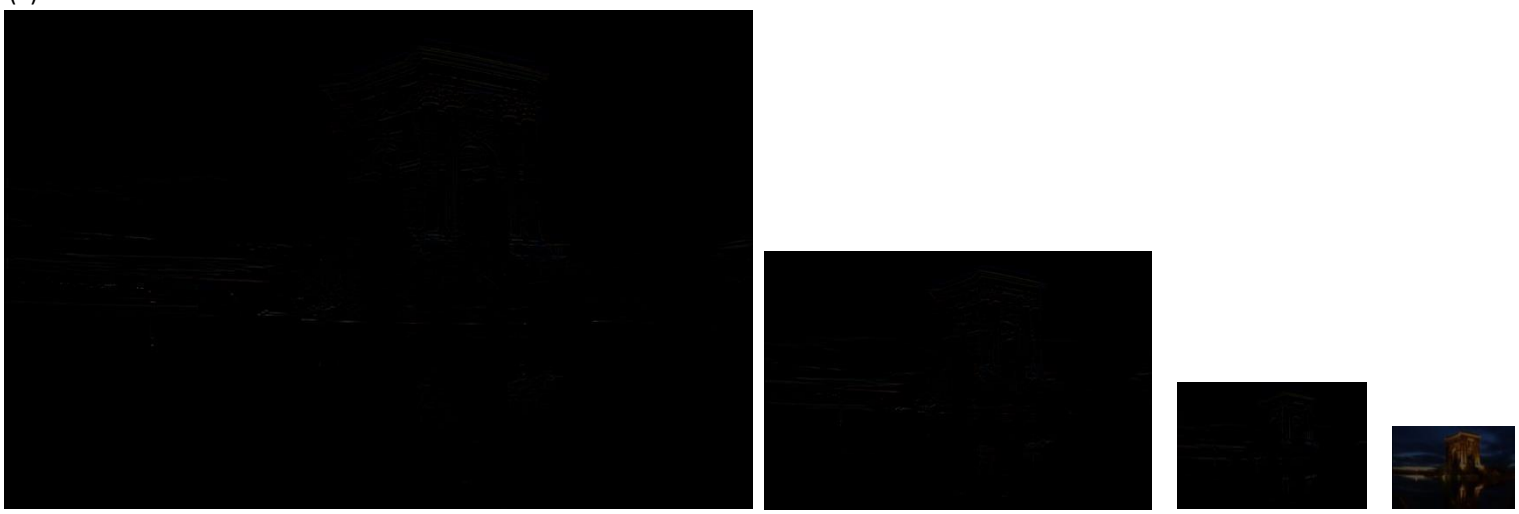
We now obtain the images resulting from the fusion of the single levels of the Gaussian and Laplanic pyramids, attenuated through the "cv2.multiply" function.

(1)



1200x800 600x400 300x200 150x100

(2)

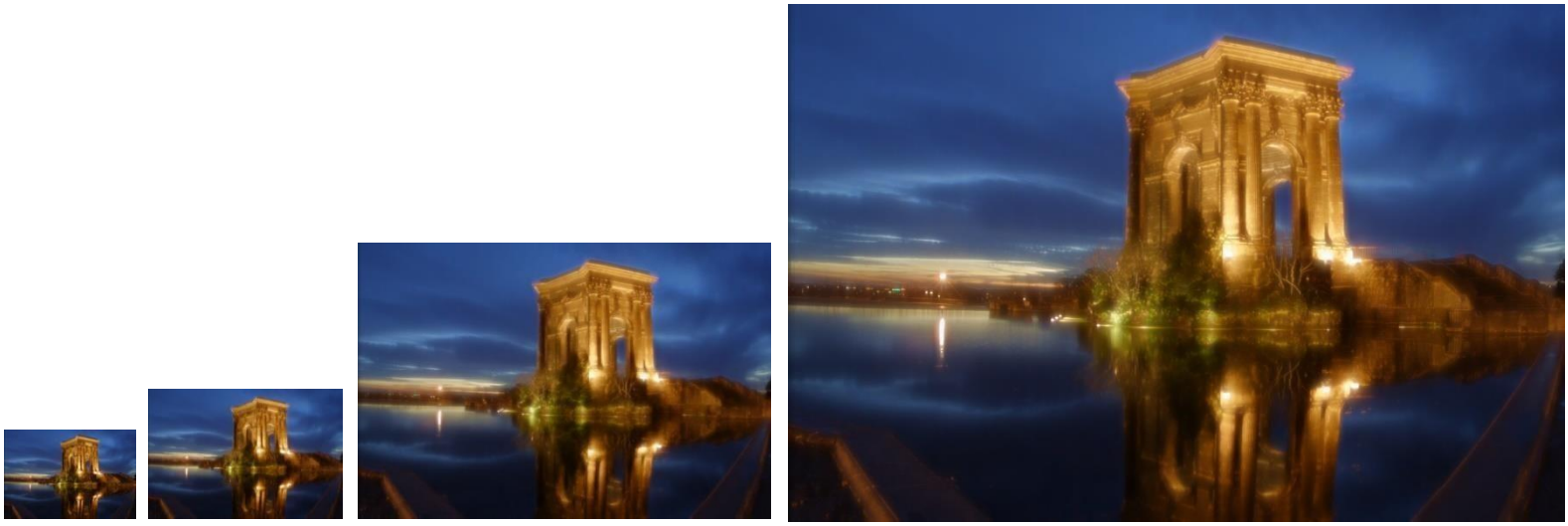


1200x800 600x400 300x200 150x100

(3)

1200x800 600x400 300x200 150x100

Now let's add up the various pyramids obtained to obtain the final image, before adding them we apply the "lowPassFilter" function to resize them correctly

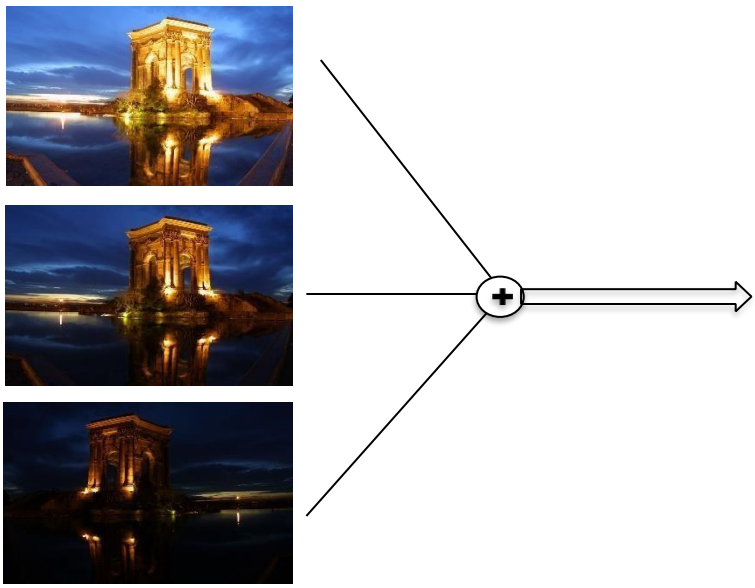


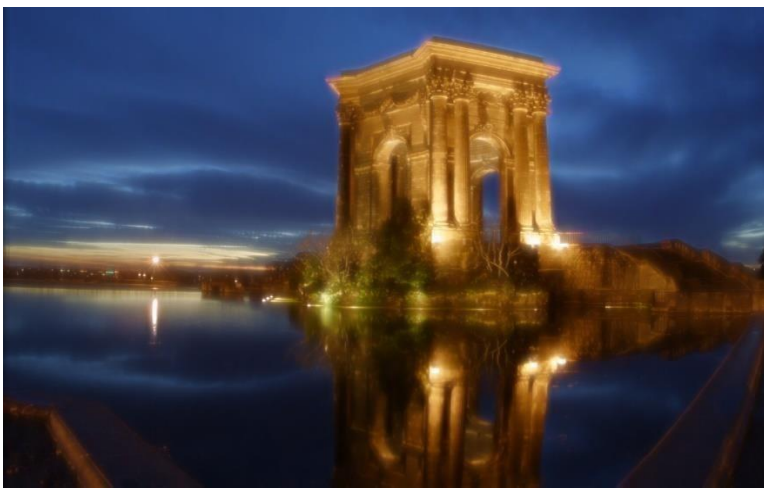
150x100 300x200 600x400 1200x800

The image obtained and then returned by the algorithm will be the last of the original resolution 1200x800.

Final results

Monument





Using 4 as the level number

Immediately below we find other examples of final results starting from 3 input images

Home

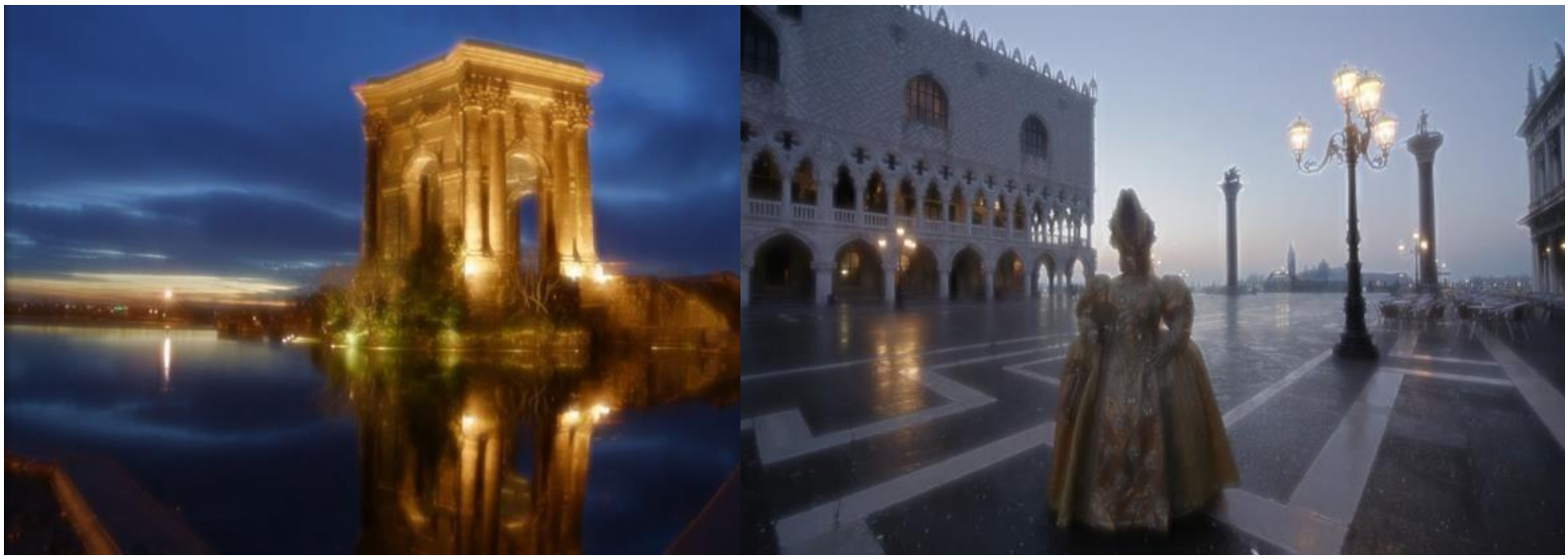
Venice

To correctly view the final results performed on various test images, follow the document Tests in the Images folder which explains where to find the images generated by the various execution phases and how to run the algorithm tests.

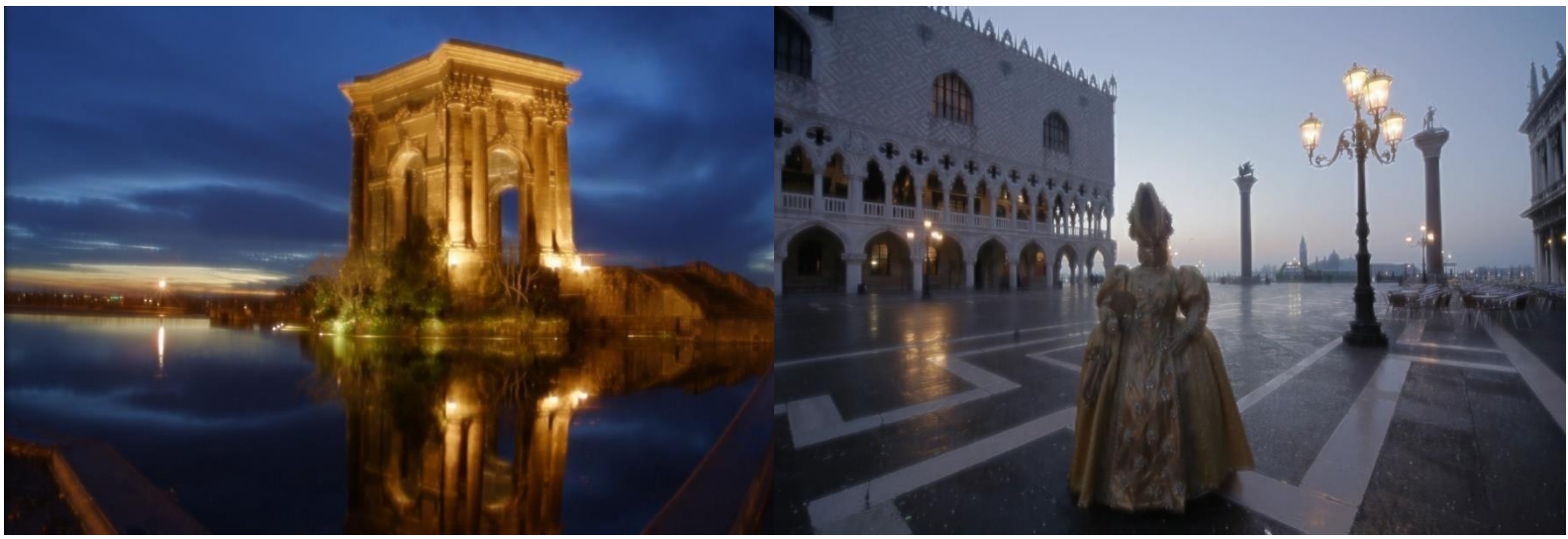
Final considerations on the results

We can see how the final result is satisfactory, but we have a particular "blur" effect, caused by the different resizes and the various overlays made during execution, which increases as the number of layers the original images are decomposed increases. Now let's see examples of execution with a different number of levels noting the change in quality.

Number of levels = 4



Number of levels = 3



Number of levels = 2