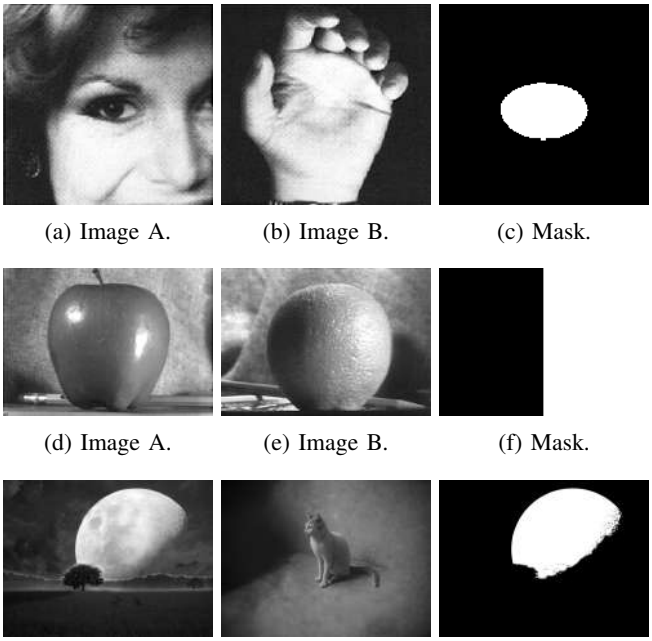
MO446 Computer Vision: Project 1

Juan Hernández
RA: 163128
Email: juan.albarracin@ic.unicamp.br

Miguel Rodriguez
RA: 192744
Email: m.rodriguezs1990@gmail.com



(a) Image A. (b) Image B. (c) Mask.

(d) Image A. (e) Image B. (f) Mask.

(g) Image A. (h) Image B. (i) Mask.

Fig. 1: Images used to make spatial blending and frequency experiments. The blending is performed by composing image A and image B through the Mask. Each row corresponds to a different experiment.

I. Introduction

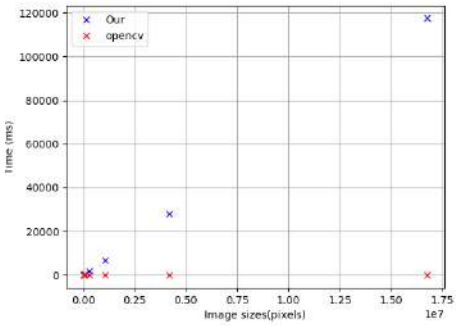
The present work shows the experimentation to perform a blending of two images through a mask, both in the spatial and the frequency domain. In order to perform this task, it is necessary to use computer vision tools such as convolution, Gaussian pyramid, Laplacian pyramid, Fourier transformation, etc. Using the images shown in Fig. 1, we will perform three blending experiments Where each experiment will be performed using two images and one mask shown in each row of figure.

The blending technique consists in making the construction of a Laplacian pyramid for both images and the construction of a Gaussian pyramid for the mask. The next step is create a new pyramid from the weighted union of each floor of the pyramids of the images, this weighted union is realized by means of the mask, which contains the form of the region where it is desired perform the blending. The final step is the collapse of the new pyramid from the smallest to the largest level, the final result is the blending image.

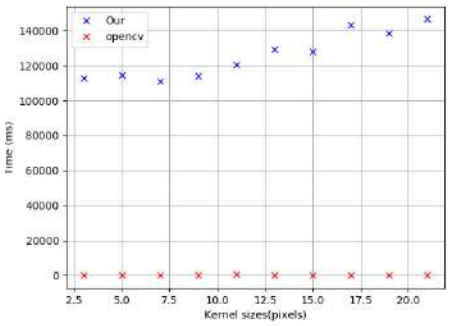
This work is divided in different experiments, each one of them is realized to test the basic tools created for doing blending. The first experiment that will be carried out will be the implementation of convolution function and we make a benchmark with the implemented function in OpenCV library; then we perform experiments with the Gaussian and Laplacian pyramids; with the basic tools developed we can perform experiments with spatial blending; then we will perform experiments in the frequency and finally a frequency blending.

II. Convolution Benchmark

The construction of the Gaussian and Laplacian pyramids is done through the use of the 2D convolution, this function



(a) Benchmark of convolution with different image sizes.



(b) Benchmark of convolution with different filter sizes.

Fig. 2: Graphs of benchmarks made between ours convolution function and the function implemented by OpenCV.

allows smoothing process on the images, so we have decided to implement out own version, which can be seen in Listing 1.

```
    mask = np.flipud(np.fliplr(mask))

    for c in range(channels):
      for x in range(cols_pad, max_cols):
        for y in range(rows_pad, max_rows):
          roi = src_img[y - rows_pad:y + rows_pad
+ 1,  x - cols_pad:x + cols_pad + 1, c]
          sum = (roi * mask).sum()
          dst_img[y - rows_pad, x - cols_pad, c] =
 sum

```

Listing 1: Extract of convolution function

In order to compare the response time of out function, we decided perform two comparative benchmarks with the implemented function in OpenCV. The first one consists in a comparison of the execution time of the functions when use images with different sizes. The resulting graph can be seen in Fig 2a, where we can see that the execution time of our algorithm is always more greater that the function implemented in OpenCV, this because this function is programmed in a low level language and it is optimized to have the best performance.

The second benchmark realized is a comparison of the execution time of the algorithms when the size of the filter changes (eg. 3x3, 5x5, 7x7, etc). As in the image size benchmark, the execution time of our implementation is much higher (See Fig. 2b), due to the lack of optimization of the algorithm. In both comparisons we could say that the time of the implementation of OpenCV is almost linear with respect to the time of our implementation.

When performing experiments changing the filter type (like high pass, low pass or band pass), the execution time of out implementation does not vary, this because the convolution function is independent of the configuration inside the mask, so that only changes the visual result when applying different filters.

The convolution function is a highly parameterizable function, The most important parameter is the way in which the convolution is to be handled at the edges of the image. One approach is make the convolution function do not use the edges, so the resulting images would be smalled than the original image. Other approach is to make the function support the convolution of the edges (the resulting image is the same size as the original), This is archived by having the index outside the edges reflect the index within the edge or have value of 0. For purposes of this work we chose to use the approach where the resulting image has the same size of the original image, because for a better implementation of pyramids and its respective reconstruction it is better not to deal with problems of padding. For treatment when it is outside the edge is to use a reflection of the pixel that is inside.

III. Pyramid Experiments

In order to represent the pyramids we decided to create an abstract class which contains a list of all images that the pyramid contains, and also contains the size of the pyramid.

The Gaussian class and Laplacian class inherit from this abstract class, in which they differ only in the construction

(a) Visual representation of Gaussian pyramid.

(b) Visual representation of Laplacian pyramid.

Fig. 3: Representation of pyramids obtained from process Fig 1b.

function of the pyramids.The Gaussian pyramid is constructed from a series of iterations where each iteration the previous floor of the pyramid is processed by the PyUp function shown in the Listing 2, this function consists in performing a convolution and then a down sampling using the downsamplingX2 function shown in Listing 3, the result of this operation is saved as a new pyramid value. This process is repeated until it meets the number of levels entered into the pyramid.

```
def PyUp(image, mask):
    up_level = con.convolution(image, mask)
    up_level = utils.scale.downsamplingX2(up_level
)
    return up_level

```

Listing 2: Pyramid up function

```
    def downsamplingX2(img):
    return img[::2,::2]

```

Listing 3: Downsampling function

The Laplacian pyramid consists of a set of images that contain the edges of the image at different levels. This structure is constructed from the Gaussian pyramid, where each level of the Laplacian pyramid is created from the subtraction of the current level of the Gaussian with the next level after applying the PyDown function (See Listing 4), a function responsible for bilinear interpolation of the image by means of the bilinear function implemented in the Listing 5.

```
def PyDown(image, shape):
    return utils.scale.bilinearX2(image, shape)
```

Listing 4: Pyramid down function

The up-sampling was done with bi-linear interpolation, as suggested in the enunciate. We implemented a function optimized to scale always to the double of the size. Since the size of the images in the pyramid should match, and we do not know whether the image on the lower had even or odd dimensions, it was necessary to specify the shape of the lower level, to up-sample exactly to the required dimensions.

```
def bilinearX2(img, sh):

  assert(np.ceil(float(sh[0])/2)==img.shape[0])
  assert(np.ceil(float(sh[1])/2)==img.shape[1])

  img_=np.zeros(sh,dtype=np.float32)

  for i in range(0, img_.shape[0], 2):
      for j in range(0, img_.shape[1], 2):
          img_[i,j] = img[int(i/2), int(j/2)]
          if i > 0:
              img_[i-1,j]=(img_[i,j]+img_[i-2,j])/2
          if j > 0:
              img_[i,j-1]=(img_[i,j]+img_[i,j-2])/2
          if i > 0 and j > 0:
              img_[i-1,j-1]=(img_[i,j-1]+img_[i-2,j
    -1])/2

  if img_.shape[1] % 2 == 0:
      for i in range(img_.shape[0]):
          img_[i,img_.shape[1]-1]=img_[i,img_.shape
    [1]-2]

  if img_.shape[0] % 2 == 0:
      for i in range(img_.shape[1]):
          img_[img_.shape[0]-1,i]=img_[img_.shape
    [0]-2, i]

  return img_
```

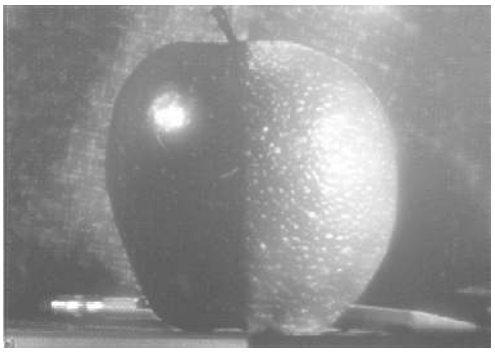
Listing 5: Bilinear interpolation function

In Fig. 3 we can observe the visual representation of the Gaussian and Laplacian pyramid. Each image is sorted by the smallest to the largest. In the Gaussian pyramid it can be observed that while more smaller the images more blurred. In the Laplacian pyramid, we can see that it is a pyramid that keep the edges of the original image, so it can be seen as a binary image in the graphic representation displayed.

IV. Spatial Blending

This section aims to perform the blending between two images through a mask, the images proposed for these experiments can be seen in Fig 1. The blending between two images A and B can be done by $blend_image$ function, shown in the Listing 6, the problem of performing this blending is that when done in the way described above the point union between both images look unnatural. In order to make the blending appear more natural, it is necessary use the pyramids described in the Section III. The algorithm consists of creating a laplacian pyramid for both images and a gaussian pyramid for the mask, and for each level to use the function $blend_img$ previously

(a) Blending experiment 1.



(b) Blending experiment 2.

(c) Blending experiment 3.

Fig. 4: Results of the blending experiments performed with Fig. 1b.

described, the result of this process is a new pyramid, which is the mixture between both images with respect to the mask. Then to obtain the final image it is only necessary to collapse the pyramid from the highest level to the base, using the inverse algorithm to that used to construct the Laplacian pyramid. This described method can be seen implemented in the Listing 7.

```
def blend_img(img1, img2, mask):
    return img1 * (1 - mask) + img2 * mask
```

Listing 6: Blend function

```
def pyramid_blending(Py1, Py2, Pym, pyramid_size):
    blends = []
    for i in Pym.inv_range():
        mask = Pym.access(i)
        img1 = Py1.access(i)
        img2 = Py2.access(i)

        blend = blend_img(img1, img2, mask)

        blends.append(blend)

      blend = blends[0]
    for i in range(1, pyramid_size+1):
        blend = cv2.add(pyramid.PyDown(blend,
  blends[i].shape), blends[i])

    return blend
```

Listing 7: Blending function

The images displayed in Fig 4 are the result of applying the pyramid blending algorithm in the different experiments shown in Fig 1. As we can see the images of the regions involved in the blending have a more natural texture than when performing the naive blending. In Fig. 4a we can see a composition where an eye was placed inside a hand, this image looks less natural, because the tone of the edge of the eye has a different skin color that the hand, and this can not be fixed by convolution, but we believe that using a smaller mask or more adjusted to the contour of the eye this blending has a better performance. The second experiment (See Fig. 4b) is the apple and orange composition, this is a classical blending experiment. We can see that it seem very natural especially in the mixture of the background and in the point where the apple changes to orange. This change seems more natural than when performing the naive blending. Finally the last composition, this is the least natural, since the mask allows us to transport the cat to the place of moon, in general this image has a very good composition, only that at the edges of the moon is left with a little noise, this is because in the original image the edges of the moon has a highest white values, and the mask made by us does not manage to completely take over the edges of the moon, so those this values still exists after the blending

V. Frequency Domain Experiments

This section describes a set of experiments that were made in order to determine the relevance of the magnitude and phase of the Discrete Fourier Transform, and what they represent. As suggested in the enunciate, experiments that consider the values of the pixels of both magnitude and phase image were performed.

The magnitude and phase images were obtained with the Opencv functions **cv2.dft** and **cv2.cartToPolar** as shown below.

```
def dft_mp(img):
    dft = cv2.dft(img,
            flags=cv2.DFT_COMPLEX_OUTPUT)
    return cv2.cartToPolar(dft[:,:,0],
                           dft[:,:,1])
```

Listing 8: From imaginary to magnitude/phase

The sampling criteria is as follows, depending on whether the order is increasing or decreasing (variable dec):

```
if dec:
    target = target * (np.resize(rankdata(dft_msk),
    dft_msk.shape) >= int(round(dft_msk.size * (1 -
    threshold))))
else:
    target = target * (np.resize(rankdata(dft_msk),
    dft_msk.shape) <= int(round(dft_msk.size *
    threshold)))
```

Listing 9: Sampling criteria

A. Phase experiments

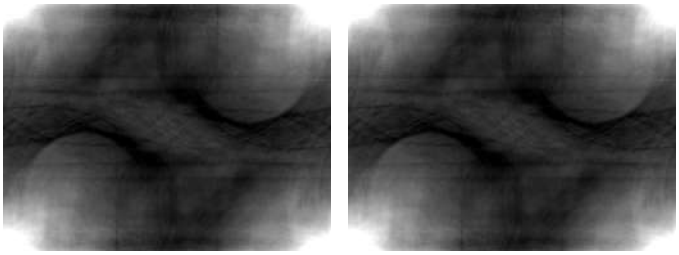
The experiments performed showed that the phase of the Fourier transform is the most sensitive, since small changes in it result in significant changes when mapped back to the spatial domain. This is because the relevant information about the structure of the image are spread in all the harmonics with different phases. Fig. *fig:fourier-experiments-phase* shows how the reconstruction of the images look like, after zeroing the harmonics with either the highest (left column) and the lowest (right column) phases. For the sake of simplicity, the images at 100% were omitted, since they look like the original image (Figure 1g).

Since the phase image contains values between 0 and π, before doing the ranking of the pixels with the highest phases, the values were shifted by $-\pi$ and then the absolute value was calculated, so shifts closer to π and $-\pi$ were considered as the most pronounced. Without doing this shift, the reconstructed images did not look as complementary as those of Fig. 5 when compared the two images at each row; they rather looked the same, but in different intensities, and that makes sense, since each image of the same row would be taking half of the information of the same shift from the origin.

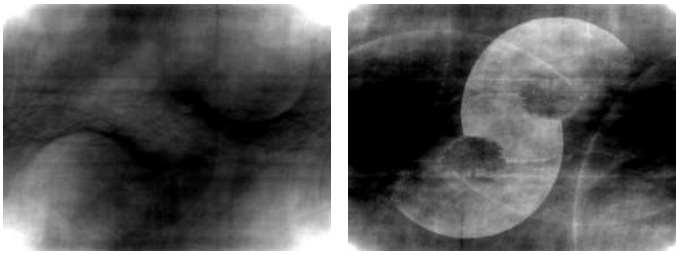
The first observation of the experiments is that, as it was expected, the relevant information is not present in one phase. The images reconstructed with only the highest and lowest value of the phase look the same, just like an image reconstructed with a zero phase. The structure of the image becomes visible when a considerable quantity of harmonics have different shifts.

Although the information is spread throughout all the phases, for the example image, there is a small shift to the side of the smallest phases, because the reconstructions of higher quality where in that range.

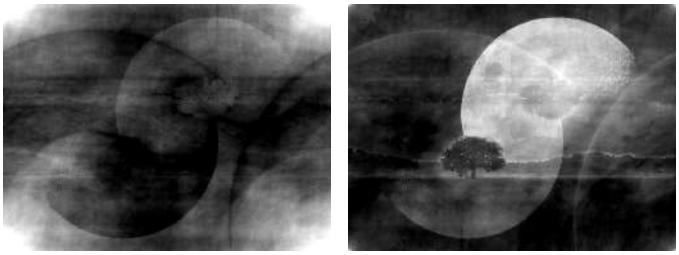
Now, the patterns observed in the reconstructed images are repetitions of the structures found in the original image. By zeroing values from either the phase or the magnitude image,



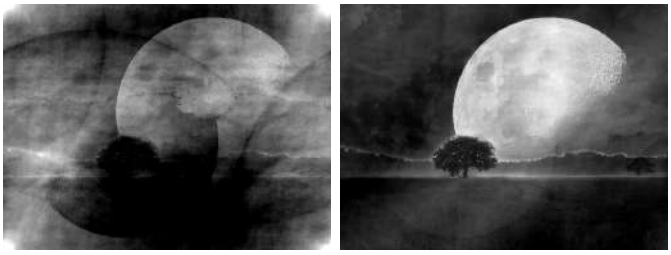
(a) Greatest point (b) Smallest point



(c) Greatest 25% (d) Smallest 25%



(e) Greatest 50% (f) Smallest 50%



(g) Greatest 75% (h) Smallest 75%

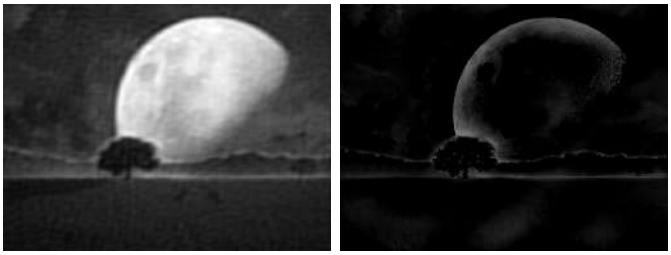
Fig. 5: Reconstructions from sampling in the phase image.

we are losing information; in phase, zeroing means aligning the harmonics, so they have the same origin, and these shifts are reflected in the spatial domain.

B. Magnitude experiments

The results for the experiments on the magnitude spectrum were more homogeneous. The suggested thresholds for the sampling of the points with the highest and lowest magnitudes yielded mostly redundant results: the reconstructed images with 25% of the highest pixel values already yielded a reconstruction that is visually the same as the original image, while 75% of the lowest pixel values obtained a black image with some light points, and the rest where almost black. Likewise the phase experiments, only one pixel in the magnitude image

(a) 1% greatest points (b) 99.99% smallest points



(c) 0.1% greatest points (d) 99.999% smallest points

Fig. 6: Reconstructions from sampling in the magnitude image.

(either the highest- or lowest-valued) yields a result practically equivalent to zero intensity.

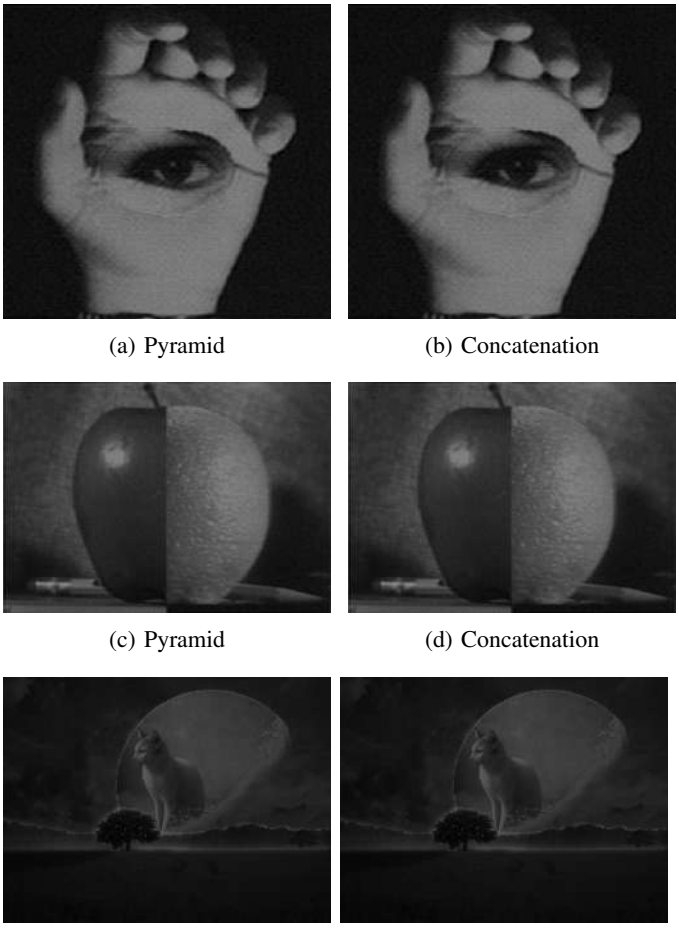
Interesting results can be visualized when working with narrower thresholds. Since 25% of the highest-valued pixels already attain the original image, we lowered the threshold progressively, until finding a different result. Fig. 6 shows, in the left column, the reconstruction using only 1% and 0.1% of the highest values and, in the right column, the reconstruction using 99.99% and 99.999% of the lowest values. These thresholds that we report here were chosen ad hoc, i.e., we lowered the portion of the highest values to select, until some reconstruction loss was visualized, and augmented the portion of the lowest values to select, until some light pixels in the image resemble the structure of the original.

VI. Relevance of both components

It can be seen that both components are relevant, depending on the application they are applied for. The fact that changes on the phase alters the structure of the image shows that the shifts of the harmonics define the distribution on the pixel values more than the amplitude. On the other side, the magnitude encode information of the intensity more than the phase. The "sensitivity" of the phase image is because the relevant information is present in various harmonics with different phases, while it is more concentrated in the harmonics with the highest amplitude. A good application for this kind of transformation is data compression, for which the magnitude seems to be the most relevant, since most of the information necessary to reconstruct reliably the image is concentrated in the frequencies of higher amplitude.

VII. Frequency Blending

For the blending in frequency, some different approaches were considered, but only two obtained results that are worth to mention in this report. The first one attends the suggestions



(a) Pyramid (b) Concatenation

(c) Pyramid (d) Concatenation

(e) Pyramid (f) Concatenation

Fig. 7: Frequency blending results using two approaches.

made in the enunciate: first, it masks the images in spatial domain, maps them into the frequency domain, takes the magnitude and phase spectrum and then blends separately the magnitudes and the phases, using the implementation we made of the pyramids for the spatial domain. Finally, the blended spectra are mapped back to the spatial domain.

```
def freq_pyramid_blending(img1, img2, mask,
    pyramid_size, kernel):
    img1_=img1*mask
    img2_=img2*(1-mask)

    dft_m1,dft_p1=dft.dft_mp(img1_)
    dft_m2,dft_p2=dft.dft_mp(img2_)
    freq_mask = 0.5 * np.ones(dft_m1.shape)

    blend_m=blending(dft_m2,dft_m1,freq_mask,
    pyramid_size,kernel)
    blend_p=blending(dft_p2,dft_p1,freq_mask,
    pyramid_size,kernel)

    return dft.idft_mp(blend_m, blend_p)
```

Listing 10: Pyramid-based frequency blending

In initial experiments, we considered to pass the Fourier transform of the mask to the pyramid blending, but in the end, the best results were attained by passing an array of 0.5, so the magnitude and phase spectra were linearly combined. Different values were tested, but they made one of the images to predominate in the result. 0.5 attained the most fair combination. The first column of Fig. 7 shows the results obtained with this approach.

Fig. 8: Resulting image of concatenation-based frequency blending before down-sampling.

The second approach consists in masking the images, calculating its phase and magnitude spectra, sub-sampling them using the pipeline described in Listing 9, concatenating vertically the two magnitude spectra and the two phase spectra separately, so the resulting images have twice their heights, and then mapping back to the spatial domain. An example of the resulting image in the spatial domain after the last mapping is shown in Fig. 8. It can be seen that the blending was (almost surprisingly) done, but, interleaved, there were rows that could be easily ignored to obtain the blended image.

```
def freq_concat_blending(img1, img2, mask):

    img1_=img1*mask
    img2_=img2*(1-mask)

    dft_m1, dft_p1=dft.dft_mp(img1_)
    dft_m2, dft_p2=dft.dft_mp(img2_)

    dft_m1,dft_p1=dft.filter(dft_m1,dft_p1,0.05,True
    ,False)
    dft_m2,dft_p2=dft.filter(dft_m2,dft_p2,0.05,True
    ,False)

    blend_m=np.concatenate((dft_m1,dft_m2),axis=0)
    blend_p=np.concatenate((dft_p1,dft_p2),axis=0)

```

```
    return dft.idft_mp(blend_m, blend_p)[::2,:]
```

Listing 11: Concatenation-based frequency blending

The sub-sampling of each image in the frequency domain consisted in keeping the highest 5% values of the magnitude image. This helped to blur the borders a little, including the transitions from one image to another. When we performed the concatenation along the horizontal axis, the results obtained where intelligible.

A. *Discussion on frequency blending*

The results obtained in this set of experiments were not as satisfactory as the results obtained in the spatial domain blending. Although the frequency domain allowed us to smoothen a little the transitions between one image and the other, the visual effect resembles a simple composition of both images. Some interesting conclusions about these experiments is that it is possible to make many operations in the frequency domain that involve strictly spatial properties. Also, the lack of documentation in the web for us to have insights on which ideas implement, made us to try many approaches that in the end obtained poor results. Maybe the most counter-intuitive fact learned was how the concatenation of the spectra yielded the blending, but this concatenation only worked when done vertically. The fact of physically joining the low-frequency spectra of the images could yield some coherence in the reconstruction. Finally, it is not so clear whether the results obtained for the pyramid-based blending in the spatial domain can be reproduced in the frequency domain: the pyramid involves linear operations that could be translated to the frequency domain, such as addition, multiplication and convolution, but up- and down-sampling are operations that are not clearly conjugated in the frequency domain, or at least they are not so obvious.