Project 1

(191079) Nathana Facion
(192803) Rafael Mariottini Tomazela

Universidade Estadual de Campinas
Instituto de Computação

Prof. Adín Ramírez Rivera

August 2017

Contents

1 Spatial Blending **5**

1.1 Convolutions 5

1.2 Gaussian Pyramid 7

1.3 Laplacian Pyramid 8

1.4 Blending 10

2 Frequency Blending **17**

2.1 Exploring Fourier Space 17

2.2 Blending 22

Bibliography **29**

Chapter 1

Spatial Blending

1.1 Convolutions

You need to create a convolution function that receives an input image and a convolution mask, and that returns the convolved image as output. A suggested signature1 for the function is

```
void convolution(InputArray input, OutputArray output, InputArray mask).
```

Your convolution function should perform a general convolution operation, and return an image of the same size as input (that is, make no assumptions about mask). In your report explain your approach to handle boarders, and justify it. Additionally, make a couple of experiments with some masks (for example test 3 3, 7 7, 15 15 masks) and measure the execution time on your function. Repeat the experiment with the convolution function provided by OpenCV. In your report, show and explain your findings, and justify your results. There is no need to show all the images (a couple of them will suffice), instead show a graph or table with the comparison and explain your findings.

Figure 1.1: Convolution with filter 3 x 3.

Figure 1.2: Convolution with filter 3 x 3 and Gaussian.

Figure 1.3: Convolution with filter 7 x 7 and Gaussian.

Figure 1.4: Convolution with filter 15 x 15.

Filter	Time Convolution(ms)	Time Convolution OpenCV(ms)
3x3	1577.198	0.703
7x7	1664.747	1.610
15 x 15	1808.997	4.343

Explain : For the test, a filter was applied with size : 3x3, 7x7 and 15x15. Using an Implemented Convolution and an OpenCV Convolution we can see that with the bigger gaussian filter applied, more details disappear from the image. About the difference between the two convolutions, an OpenCV is much faster because it presents optimized forms of programming, but the final result was the same.

For the borders, we simply repeated the edge. We think that's a good idea when using filters, specially a gaussian one as we did, because a constant value, like 0, would drag the value of the pixels close to it to that constant value. A wrap would make the pixels closer to the one in the other side. Considering that, repeating the edge looked like a good idea, since it would maintain the mean of the pixels close when adding the edge.

1.2 Gaussian Pyramid

You need to create a data structure to handle your Gaussian pyramid. This pyramid is constructed using an input image, and a number of levels. The construction process is as follows. Blur the image of a given level using a Gaussian mask (explain your choice of mask size or experiment with different sets of masks and show your results), and then down-sample it by removing each other row and column. The image of the next level is the result of the previous operation. Repeat this process for all the levels. The first image is the original input image. Note that since you are removing every other row and column during the down-sampling operation, you are reducing the size of the image by a factor of two. For the reconstruction process of a given level, you need to perform the inverse operation. First, you duplicate the size of the level and insert the existing pixels in every other row and column. The missing information should be interpolated. You can use bilinear interpolation (if you want you can test other types of interpolation and explain them in your report). Your data structure should implement these two operations with two functions: up and down, that receive an image and return the corresponding image. Note that since we define our original image to be the base level of the pyramid, we will construct it using the up function, while we will recover the previous images by using the down function. Additionally, add an access function to obtain a given level of your pyramid when needed. Explain your design decisions on the implementation of the data structure and reason why.

Figure 1.5: Access level 0

Figure 1.6: Access level 1

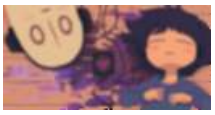


Figure 1.7: Access level 2

Figure 1.8: Access level 3

Figure 1.9: Access level 4

Explain : A GaussianPyramid class has been created. A GaussianPyramid class has the following methods: up, down and access. As was suggested in the project description. The class receives in its constructor an image and the number of levels the pyramid should have, and then it constructs all levels and stores them in an array. The access function simply receives a level and returns the corresponding level from the array.

We use the up function to create all the image levels in the class constructor. It's also possible to call the up and down function externally, so if you call the down function for a certain level X, it will upsample and interpolate the image to return one with double the width and height.

The pyramid class was designed like this for some reasons. Firstly, it's useful to pre-compute all levels in the constructor and store them instead of recreating the levels every time some level is accessed. The up and down function receives the level of the image, instead of an image, because it makes sense to tie these functions to the image. These abstractions proved to be efficient and powerful for our tests.

To make things easier, we only considered the image could be in the size $2^n + 1$. We did that because it's easier to make interpolation and sampling with these sizes, specially when doing the pyramids which need to halved or doubled in size repeatedly.

1.3 Laplacian Pyramid

Similarly to the Gaussian pyramid, you will construct a Laplacian pyramid data structure with similar functionality. The same idea of up and down functions apply. The construction of a level of the Laplacian pyramid is as follows (up operation). Let G be the corresponding Gaussian pyramid, of the given image. At

level i, you need to reconstruct the next level of the Gaussian (e.g., G[i].down() or Gaussian::down(G[i]) depending on your definition of the previous section) and subtract it from the current level image (in the Laplacian pyramid). The result is the current Laplacian level. The reconstruction (down operation) of a given level does the opposite operation (that is, it reverses the operations to produce the original image). Explain in your report how do you perform the reconstruction in detail. Additionally, explain your design decisions on the implementation of the data structure and reason why.

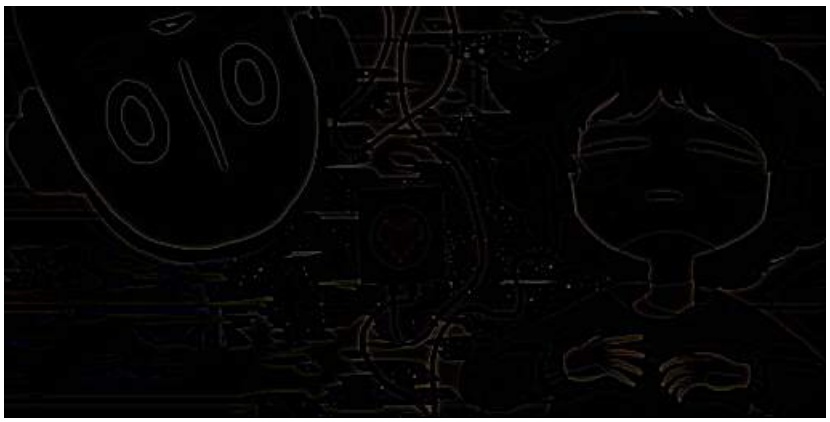


Figure 1.10: Access level 0

Figure 1.11: Access level 1

Figure 1.12: Access level 2

Figure 1.13: Access level 3

Figure 1.14: Access level 4

The down operation was done by doing billinear interpolation and doubling the width and height of the image. The up operation was done as the instructions said, but instead of downsampling a lower pyramid level, we made the up operation so it returns the level it receives as a parameters. That is, if

we are trying to do the up operation on level X, we first use the down operation on the same image for the gaussian pyramid on level $X + 1$, and then subtract it from the gaussian level X, and so we get the corresponding level X of the laplacian pyramid. By definition, as given in [1], we consider the last level of the laplacian pyramid the same as the last level of the gaussian pyramid.

To accurately reconstruct the image, following the paper [1], we decided to implement the down operations as the EXPAND operation with only one level. Using that, it is possible to use the Summation Property of the laplacian pyramid to recover the original image. That is, to reconstruct the image we started with the last level of the pyramid, expanded it (using the down operation) and then added with the previous pyramid level, to also recover the previous gaussian level. We repeated this process for all levels until we got the original image.

The design decisions of the Laplacian Pyramid were the same as for the gaussian one. The difference is that the laplacian pyramid creates a gaussian pyramid for the same image and levels so it can use in its operations. There is also the recover_original method, which recovers the original image from the levels. We could easily create a method to also recover some certain level of the gaussian pyramid, but it wouldn't be used for anything, so we decided against it to keep it simple.

1.4 Blending

Using your implementation of the Gaussian and Laplacian pyramids you need to implement the blending operation proposed by Burt and Adelson [1]. That is, you need a pair of images and a blending mask, of the same sizes. The blending mask is an image of the same size as the inputs that contains 1's (or 255's in case of 8 bit images) that marks the interest region. Construct your Gaussian and Laplacian pyramids of a given pair of images, and the Gaussian pyramid of the mask. At the highest level of the Laplacian pyramids, blend the corresponding images using the mask. The first image should use the mask as is, and the second one should use its complement. Then, reconstruct the resulting image with the down operation of the Laplacian pyramid data structure. Explain your process in your report and show your results. Try to reproduce the results presented by the original authors [1]. Test your algorithm with several images and different masks (include such images in the input folder). Explain the limitations of your algorithm and show cases in which it fails. Can you explain why?

Reproduced the results presented by the original author:

Input:

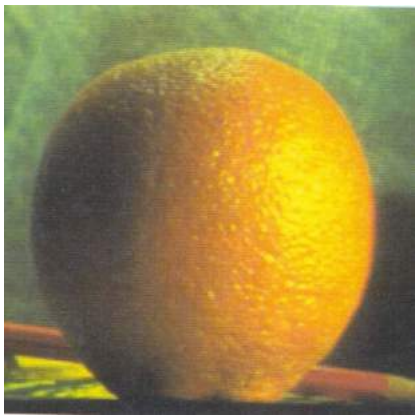


Figure 1.15: (p1-1-3) - Without mask - Apple

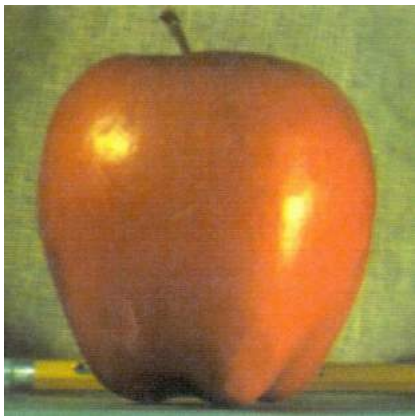


Figure 1.16: (p1-1-4) - Without mask - Orange

Output:

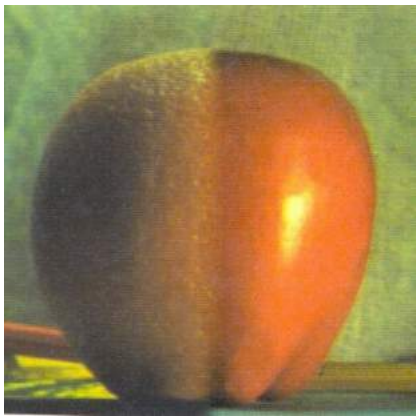


Figure 1.17: (p1-2-4-0) - Without mask - Apple + Orange

Input:

Figure 1.18: (p1-1-10) - Without mask - People

Figure 1.19: (p1-1-11) - Without mask - Hand

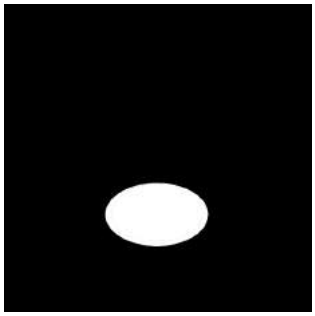


Figure 1.20: (p1-1-9) - Without mask - Oval Mask

Output:

Figure 1.21: (p1-2-4-3) - With mask of other image - Eye + Hand

More examples:

Input:

Figure 1.22: (p1-1-5) - Without mask -Vegeta

Figure 1.23: (p1-1-6) - Without mask - Goku

Output:

Figure 1.24: (p1-2-4-1) - Without mask - Goku + Vegeta

Input:

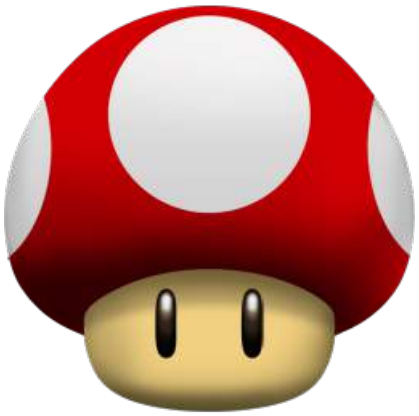


Figure 1.25: (p1-1-7) - With mask circle(code) - Red Mushroom

Figure 1.26: (p1-1-8) - With mask circle(code) - Striped Mushroom

Output:

Figure 1.27: (p1-2-4-2) - With mask circle(code) - Red Mushroom + Striped Mushroom

Firstly, we recreated the first part of the paper where it doesn't use any mask, but only divide the image in the middle. We did that so we could understand better what was happening. For that part, we tried to reproduce the apple and orange experiment as shown in 1.17, and also another example with a higher contrast cartoon image as shown in 1.24.

We then tried to reproduce again the experiment using masks. Since we didn't find the original one, we used another hand and eye combination, as shown in 1.21. We also tried again with two cartoonish images as shown in 1.27, but we used a circular mask that we defined in code.

One of the limitations of our algorithm is the fact that input images and mask must have the exact same width and height and their sizes must be $2^n + 1$. As we said before, it was done to make it easier to construct the pyramids. As we can see, when there is high contrast such as in 1.24, the algorithm does not make a really smooth transition. To make the transitions smoother we would probably also need to detect similar points, and make transformations so the hair and clothes could combine in the same spots.

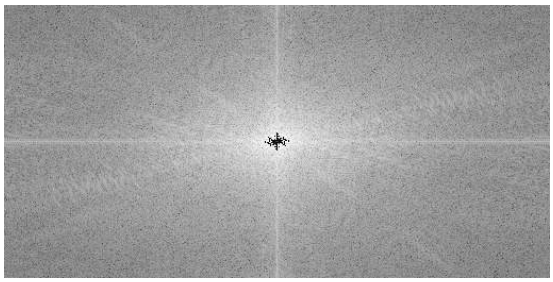
Chapter 2

Frequency Blending

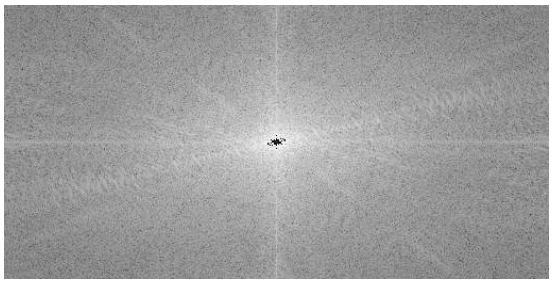
2.1 Exploring Fourier Space

Before doing the blending you will perform some experiments with the frequency domain. Pick an image and convert it to the frequency domain (you can use OpenCV Fourier transform operations for this). You should have a magnitude and a phase image. Select the phase image and select the lowest value (bigger than zero) in that image. Create a new phase image that contains only the selected pixels. And use it to reconstruct an image. Save your result. Reproduce this step, with the difference that you add more points per iteration. Show your results at the 25%*,* 50%*,* 75%*, and* 100% *of selected points. You should have five images (one with one point, and four with the given percentages of points). Repeat this process with the phase image, but select the points in decreasing order. That is, select first the highest frequency, and then include the percentages of the top* 25%*, and so on. You should have five images as well. Repeat the same process of decreasing and increasing orders and selection for the magnitude image. I advice you to create a function that builds this functionality given an image and a number or percentage of points to include.*

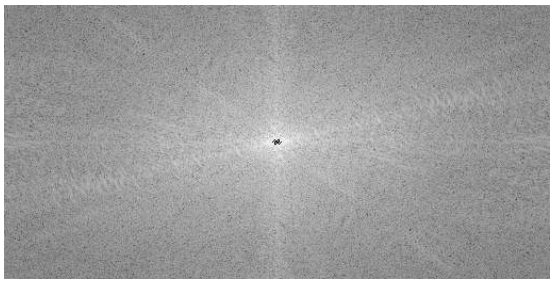
Show and explain in your report what your results are. Does the magnitude and the phase have the same importance? Explain your answer.



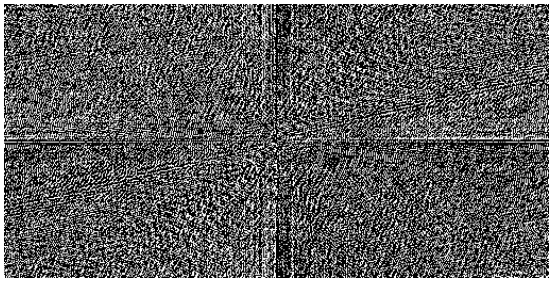
(a) (p1-3-1-0) - magnitude R



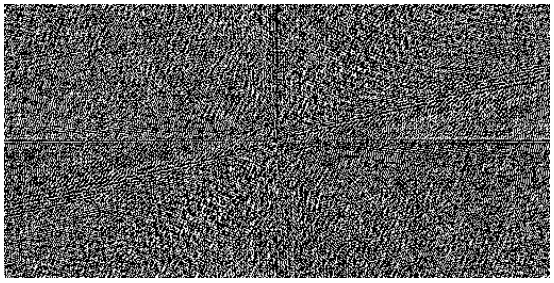
(b) (p1-3-1-1) - magnitude G



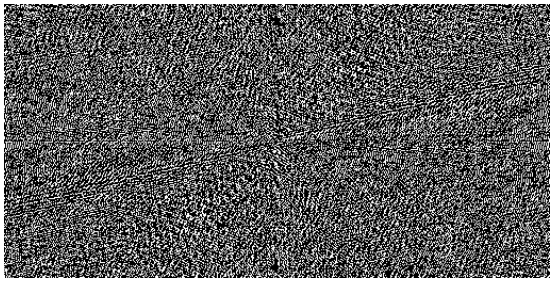
(c) (p1-3-1-2) - magnitude B



(d) (p1-3-1-3) - phase R



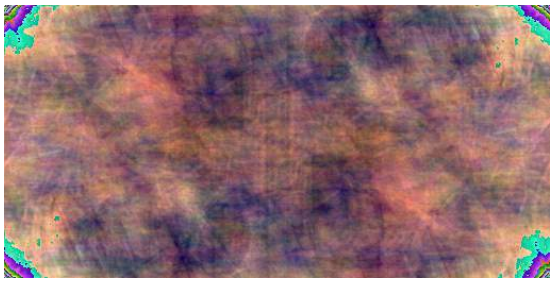
(e) (p1-3-1-4) - phase G



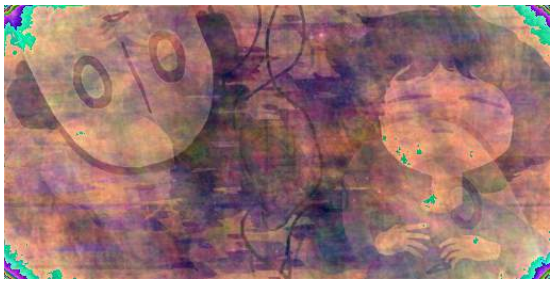
(f) (p1-3-1-5) - phase B

Figure 2.1: Magnitude and Phase - RGB

We applied a log function to the magnitudes so we could see the resulting image.



(a) (p1-3-1-6) - with min



(b) (p1-3-1-7) - 25%

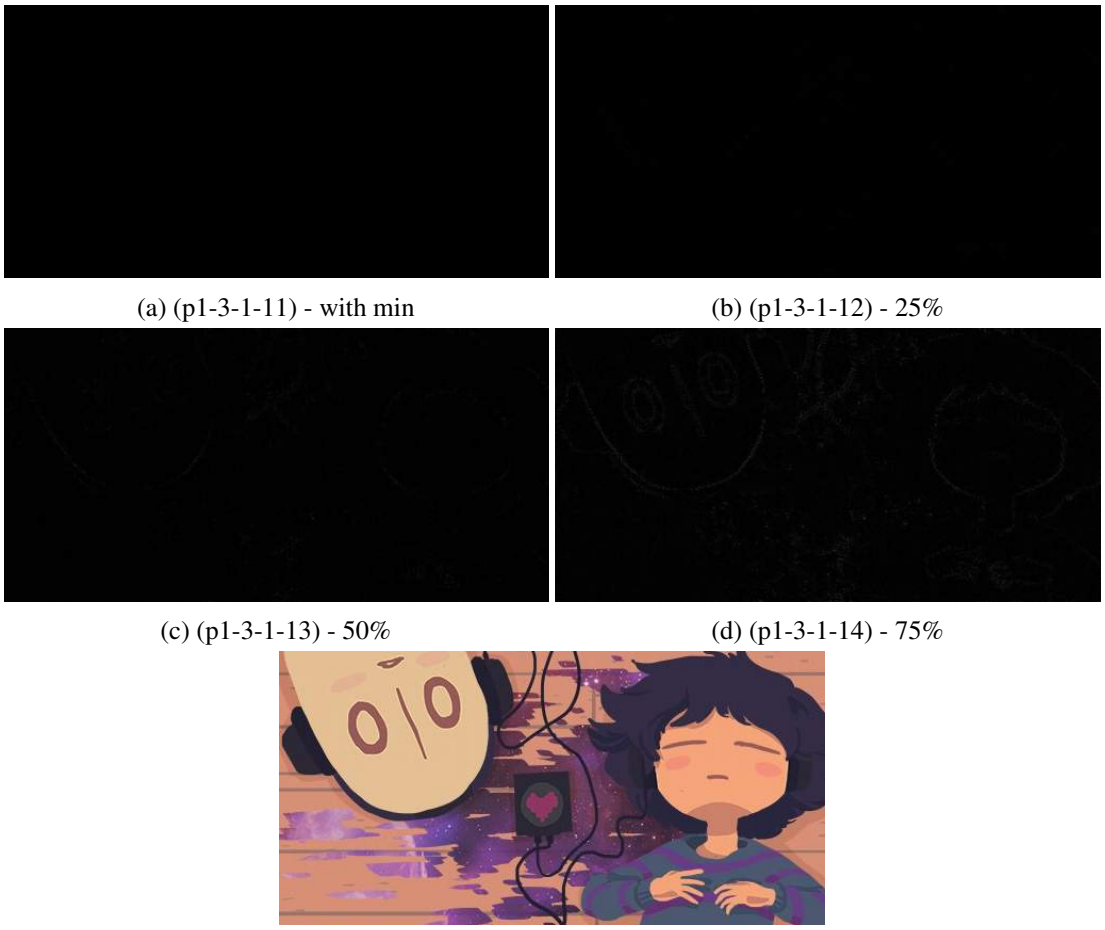
(c) (p1-3-1-8) - 50%

(d) (p1-3-1-9) - 75%

(e) (p1-3-1-10) - 100%

Figure 2.2: Alter: Percentage phase up

In the above images, with first have the resulting image with only the smallest non zero value, then with the 25% smallest and so on. As we can see, although the colors are all there since we preserved the magnitude, the image is kind of "scattered" everywhere since we do not have the phase and all the sines are summing in the wrong place. As we add more points to the phase, we can see that the image starts to look closer to the original one, and the "scattered" or "ghostery" images in the wrong places, disappear. That is because the fourier signal starts to look closer to the original one.



(a) (p1-3-1-11) - with min

(b) (p1-3-1-12) - 25%

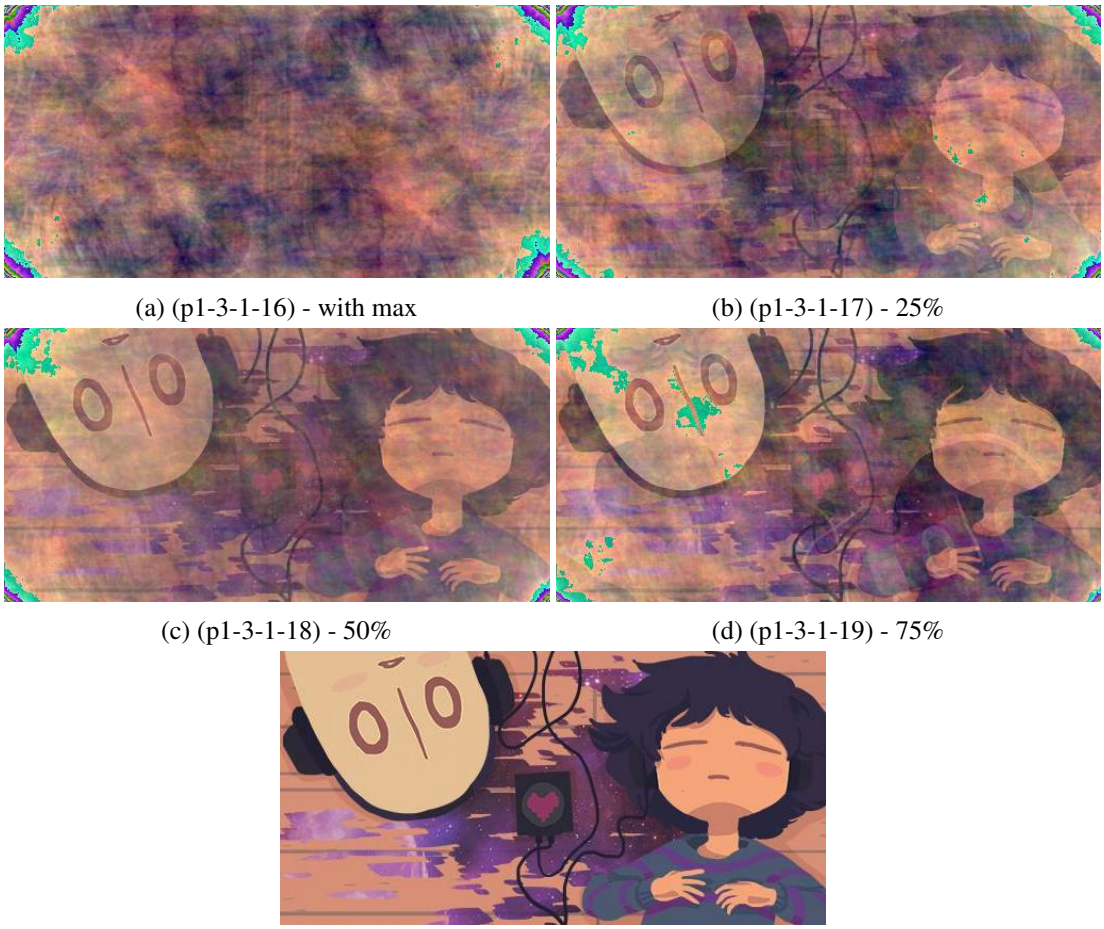
(c) (p1-3-1-13) - 50%

(d) (p1-3-1-14) - 75%

(e) (p1-3-1-15) - 100%

Figure 2.3: Alter: Percentage magnitude up

In the above images we do the same thing, but now with the magnitude. As we can see, there is close to nothing to see when we don't select the highest magnitudes. We can conclude, then, that a few magnitude points are mostly enough to have the image. That's because the sines that have the biggest amplitude are the ones that makes most of the "format" of the fourier.



(a) (p1-3-1-16) - with max

(b) (p1-3-1-17) - 25%

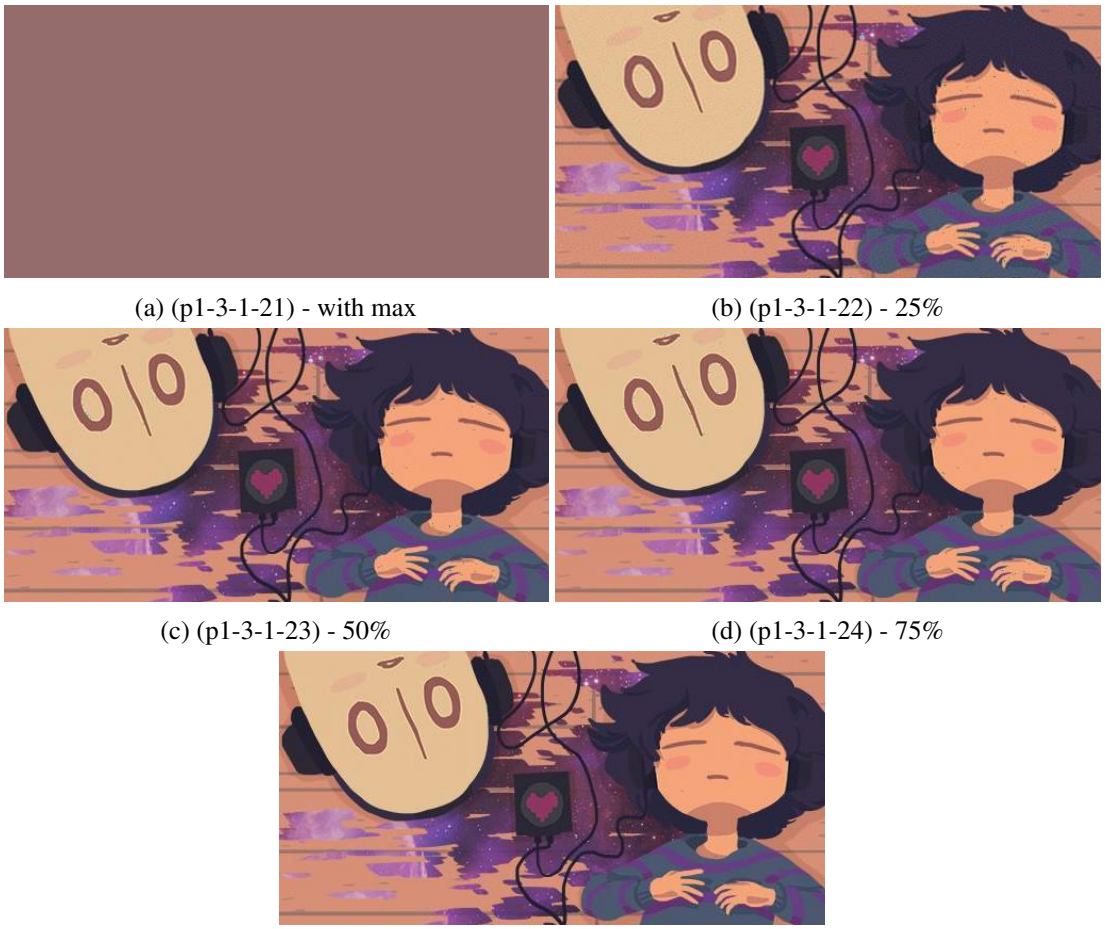
(c) (p1-3-1-18) - 50%

(d) (p1-3-1-19) - 75%

(e) (p1-3-1-20) - 100%

Figure 2.4: Alter: Percentage phase down

Since the phases are mostly angles, removing the highest or lowest values doesn't make a big difference. What is important is that we have enough sines summing in the right places to recover the original image. That's why it doesn't make a big difference to remove the highest or lowest values.



(a) (p1-3-1-21) - with max

(b) (p1-3-1-22) - 25%

(c) (p1-3-1-23) - 50%

(d) (p1-3-1-24) - 75%

(e) (p1-3-1-25) - 100%

Figure 2.5: Alter: Percentage magnitude down

As we saw before, if we remove the points with the highest magnitude, we can not see the image clearly. The converse is true if we get the points with the top magnitude. As we can see, only the points with the highest values are enough to get the general color of the image. With only 25% the image is already close to the original, with only some artifacts. The reason for that is the same: the sines with the highest amplitude are the most important ones to recover the image.

We can also see, as explained before, that the phase is way more important than the magnitude to recover the image. The magnitude will only recover the amplitude of the sines, and so only some sines with a high amplitude are enough to recover the image. But phases in the wrong place make it impossible to recover the original image, since we can't really recover anything that resembles the original signal.

2.2 Blending

Using the same images (input, output, and mask) as in 2.4*, mask the images. That is, you should have information on the corresponding part of the mask, and zero elsewhere. Note that your mask need to be inverted for one image (similar to the process of the blending in the spatial domain). Convert the masked images to the frequency domain. Now, you need to blend them together. And convert the blended frequencies into an image. Experiment with different blending methods of the phase and magnitude. Use your experience from the previous section. In your report, explain your proposed blending method, and*

why it works (or doesn't). Show results comparing the spatial blending with the frequency blending.

We merged in the frequency domain only the 1.18 and 1.19 images with the mask 1.20.

We tried a lot of different things to merge the image. In the first tries, we tried to first mask both images with the mask, and then apply a multitude of different operations in the phase and magnitude of each image to reach some result. But this proved unfruitful and almost no images were close to good.

We think that the biggest hurdle in doing these operations in the frequency domain, is that the mask blending is mostly a spatial operation. We want the area with high gradient between the two images to be smooth, but only that area. But if we work directly in the frequency domain, we can only do, easily, operations in the whole image. So if we smooth the image, we are going to smooth everything.

Also, the magnitude and phase are hard to work with. The magnitude of natural images are not so different, so it's not as important to blending. And if we worked on the phase, we caused some ghostery effects in the image. We had a somewhat better result when working with the frequencies, instead of the magnitude and phase.

Still, we had some good results as we are going to show.

Figure 2.6: (p1-3-2-1) - Simple sum

The first thing we've done is to simply sum both images, as we can see in 2.6. Since the sum of the images in the spatial domain is the same as the sum of the frequencies, the simple sum just put the two images together, as expected.

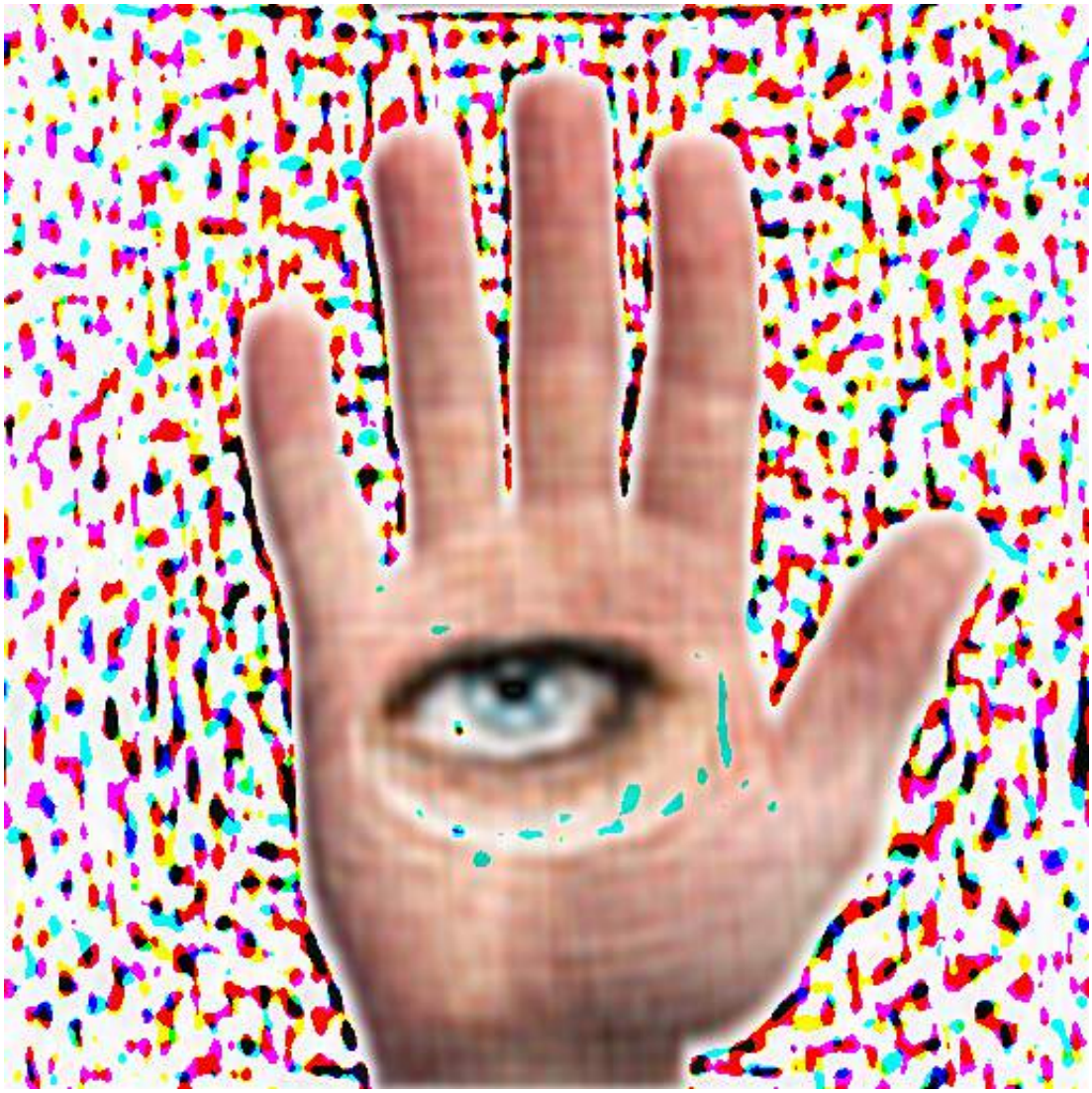


Figure 2.7: (p1-3-2-1) - Only top 1% magnitude

In the figure 2.7 we removed 99% of the lowest magnitudes of both images, and then summed them. Since we lost a lot of details, the gradient was smoother between the images. The downside is that not only we lost details, but we had many artifacts.

Figure 2.8: (p1-3-2-2) - Only lower frequencies

Instead of removing the magnitudes, we also thought of removing the higher frequencies, as shown in 2.8. So now we have a smoother image. The downside is that the image is entirely smoothed, and we have some "rings" or "ripples" in the image, which cause some artifacts to appear.

Figure 2.9: (p1-3-2-0) - Laplacian blended pyramid

What we think is our best result, which could probably turn in a more perfect image, is 2.9. For this one, we simply used the laplacian pyramid as we did before, but instead of using the images, we passed to them the fourier of the images. We then recreated all operations so they used the image frequencies instead. We only "cheated" on this when we needed to mask the images to create the mixed pyramid. The problem is that masking is mostly a spatial operation, and so it's hard to do in the frequency domain. So we would need to convolve the image and the mask, and that was a really costly operation that took a long time to finish, and we couldn't make it work in time. We think it's fair to return the images to the spatial domain when multiplying the images, because it's common to do the inverse, that is, to multiply the images in the frequency domain when you want to convolve them in the spatial domain.

Instead of sampling and doing the interpolation for the up and down functions, it was enough to crop or pad them accordingly. In the case of the up operation, it was also necessary to apply the gaussian filter (in the frequency domain) before cropping, so it wouldn't have the "rings" we saw in image 2.8.

The end image wasn't perfect, we probably messed some operation in the frequency domain. But as we said before, since the blending is mostly an spatial operation, it's not such a good idea to do it in the frequency domain. So although it's somewhat far from **??**, which is the image we wanted to recover, we

think it was a good end result since it is a difficult thing to do.

Bibliography

[1] Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Trans. Graph.*, 2(4):217–236, October 1983.