Introduction to Computer Vision – Homework 1
RA192617 – Edgar Rodolfo Quispe Condori
RA192618 – Darwin Ttito Concha
Institute of Computing, University of Campinas (UNICAMP)
Campinas-SP, Brazil, 13083-852

1 Test Images

In order to test our implementations, we have choose 4 images (figure 1), extensive experiments have been performed in each one, but more representative results are shown in this report. Complete results can be found in $output$ directory after running $make$ command. For the blending experiment, extra images are used.

The execution time of the $make$ script may take a while.

(a) Input p1-1-0.jpg, 400x400 pixels

(b) Input p1-1-1.jpg, 400x400 pixels

(c) Input p1-1-2.jpg, 300x300 pixels

(d) Input p1-1-3.jpg, 300x300 pixels

Figure 1: Images used in experiments

2 Convolution

The implementation of this function has the following signature:

```
convolve(img_input, kernel, padding_type = 'zero', padding_color = 0, normalize = False):
```

The kernels tested are shown in figure 2.

Results for each kernel are shown in figure 3. In the case of the sobel kernel, it corresponds to the vertical component (sobel filter has vertical and horizontal components), thus the results highlights the vertical lines (more clear in the wall of brick). Analyzing the pattern in the line detector it seems like an horizontal line, thus in the result of this kernel, vertical lines disappears. In the case of the Laplacian of Gaussian (LoG) it is working as an edge detector. In the media and Gaussian the results are blurred images, but in the case of the media kernel the result is bigger because of the size of the kernel.

Different kinds of padding are available: $zero, mirror$ and $constant$, the results for different padding strategies are shown in figure 4.

The zero padding suffers of white borders, this may not be logical since 0 is equivalent to black, but if we analyze the convolution kernel (laplacian of gaussian) it makes sense. Mirror padding has the best results because it does not saturate the borders with an specific color. Contrary to zero padding, the constant padding with $c = 255$ turns the borders black, the constant padding with $c = 128$ has the most similar result to mirror approach.

Because of these results, we consider mirror padding in future experiments.

-1	0	1
-2	0	2
-1	0	1

(a) Sobel 3x3

-1	-1	-1
2	2	2
-1	-1	-1

(b) Line detector 3x3

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

(c) Laplacian of Gaussian 5x5

0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

(d) Gaussian 7x7

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(e) Media 15x15

Figure 2: Kernels used in experiments

(a) Sobel

(b) Line detector

(c) Laplacian of Gaussian

(d) Gaussian

(e) Media

Figure 3: Results of convolution

(a) Zero padding (b) Mirror padding (c) Constant padding $C = 128$ (d) Constant padding $C = 255$

Figure 4: Padding results with Laplacian of Gaussian filter

The measure of the execution time of the convolution function is shown in table 1. Note that since two images are 300×300 and the other two are 400×400, we evaluate the time using one of each dimension. In the same way, we consider only one kernel of 3×3 out of the original three.

From the table we can see that the our implementation is much more slower than the OpenCV one, this is because the OpenCV library has its methods optimized, an option to improve our implementation is to use $cython$[1]

Image		***kernel***		***execution time(seconds)***		***Speedup***
name	***dimension***	***name***	***dimension***	***Our implementation***	***OpenCV***	
p1-1-1.jpg	400x400	Sobel	3x3	1.0571	0.0002	5285.5
		Laplacian of gaussian	5x5	1.1044	0.0004	2761
		Gaussian	7x7	1.1191	0.0008	1398
		Media	15x15	1.2273	0.0026	472
p1-1-2.png	300x300	Sobel	3x3	0.5934	0.0001	5934
		Laplacian of gaussian	5x5	0.6023	0.0002	3011
		Gaussian	7x7	0.6175	0.0004	1543
		Media	15x15	0.7376	0.0024	307

Table 1: Measures of the execution time

3 Gaussian Pyramid

The implementation of the Gaussian pyramid follows the next structure:

```
class gaussian_pyramid:
    pyramid = []

    # Initialize the pyramid based on an image, the number of
    # level and a kernel (Gaussian kernel is the default)

    def __init__(self, img, levels, kernel = None, gauss_kernel_par= 0.3):

    # interpolates the missing information

    def interpolation(self,x,y,v, interp = 'bilinear'):

    # upsamples an image, using the desired interpolation method

    def up_sample(self, image, size, interp = 'bilinear'):

    # downsamples an image, using the desired padding strategy

    def down_sample(self, image, size, padding_type = 'mirror',

    # Implements the down operation of the pyramid using the upsample function

    def down(self, level):

    # Implements the up operation of the pyramid using the downsample function

    def up(self, level):
```

[1] http://www.pyimagesearch.com/2017/08/28/fast-optimized-for-pixel-loops-with-opencv-and-python/ shows a solution for this problem.

```
# builts the whole pyramid using up function
def build(self):
# returns the desired level of the pyramid
def get(self, level):
# plots each pyramid level
def show(self, name = 'gauss_pyramid'):
```

We decided to use this architecture because it needs to compute each level of the pyramid just once, this is useful when function get is implement recovering each level in $O(1)$, in the same way, this implementation is more transparent for an user, allowing easy and fast use of the pyramid as shown below.

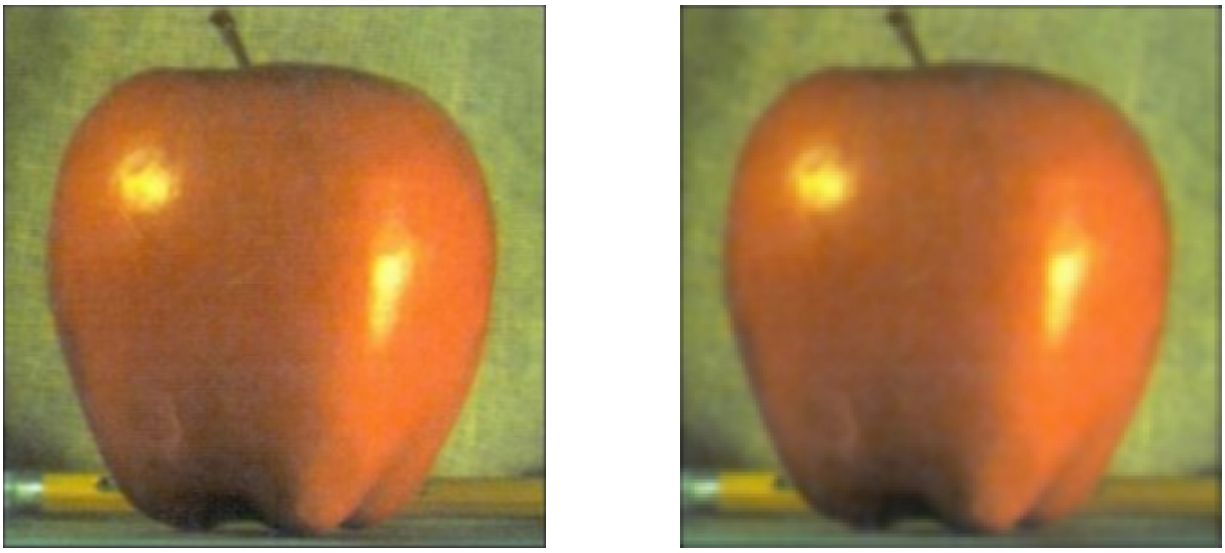
```
pyramid = gaussian_pyramid(img, levels = 7)
pyramid.build()
```

Some details of the implementation includes: it can handle multichannel and grayscale images indifferently, it implements only bilinear interpolation[2], the $down$ function may not return an image of the size of the original one whenever the height or width of the image is odd.

The results of the applying the up function to built the pyramid are shown in figure 5

Figure 5: Gaussian Pyramid

The result of the $down$ function comparing with the original pyramid level are shown in figure 6. It can be seen that the upsampled image is not equal to the original one, there is a lost of information that is not recoverable.



(a) Original image

(b) Upsampled image

Figure 6: Comparison of results of $down$ function

[2]Based on http://supercomputingblog.com/graphics/coding-bilinear-interpolation/

4 Laplacian Pyramid

Similar to Gaussian Pyramid, the implementation of the Laplacian Pyramid, follows the next model:

```
class laplacian_pyramid:
    pyramid = []

    # Initialize the pyramid based on an image and the number of levels, the kernel is define
    # for the Gaussian pyramid used to build the Laplacian Pyramid

    def __init__(self, img, levels, kernel = None, gauss_kernel_par = 0.3):

    # Addition or subtraction an image of level "i" in the
    # pyramid with the upsampled image of the level i+1

    def gauss_operation(self, gauss_cur, gauss_down, operation = '-'):

    # upsamples an image with the down function of the gaussian pyramid

    def up_sample(self, image, size):

    # upsamples an image of a level i of the laplacian pyramid and
    # adds the image of level i-1 of the laplacian pyramid, obtaining
    # the images of the level i of the gaussian pyramid

    def down(self, up_level_img, cur_level_img):

    # Implements the up operation of the pyramid using the downsample
    # function, this function allows us create one level the
    # laplacian pyramid

    def up(self, level, gaussian_pyramid):

    # builts the whole pyramid using up function

    def build(self):

    # Reconstruct the original image using the laplacian pyramid

    def reconstruct(self):

    # returns the desired level of the pyramid

    def get(self, level):

    # plots each pyramid level

    def show(self, name = 'laplace_pyramid'):
```

As in the Gaussian implementation, we choose this architecture in order to compute each level of the pyramid just once, this be useful in the blending implementation.

Some details of the implementation includes: it can handle multichannel and grayscale images indifferently, the function $gauss\_operation$ can handle the cases where the upsampled image does not have the same shape as the current level, the $down$ function up samples an image and sum it with another image, this function is the core the the $reconstruct$ function. The last level is the same in Laplacian pyramid and Gaussian pyramid.

The logic to reconstruct the image is shown in figure 7. Initially, the last level i of the Laplacian pyramid is upsampled, and then is added to the level $i-1$, the first row of figure 7 shows the elements inside the Laplacian pyramid, the second row shows the intermediate results of upsampling an image that was previously added with the corresponding Laplace pyramid level. Note that the results of the sum up (third row in figure 7) are more clean and look like the original image.

5 Blending

The function $blending$ receives two images, the mask and the pyramid levels, it returns the blended image.

```
blending(img_a, img_b, mask, level = 2)
```

Initially, the Laplacian pyramid of $image_a, image_b$ and Gaussian Pyramid of the $mask$ are build.

```
lp_a = lp.laplacian_pyramid(img_a,level)
lp_b = lp.laplacian_pyramid(img_b,level)
gp_mask = gp.gaussian_pyramid(mask,level)
lp_a.build()
lp_b.build()
gp_mask.build()
```

Next, the images of each level of Laplacian pyramids are joint using its corresponding level in Gaussian pyramid of the $mask$, they are stored in mid.

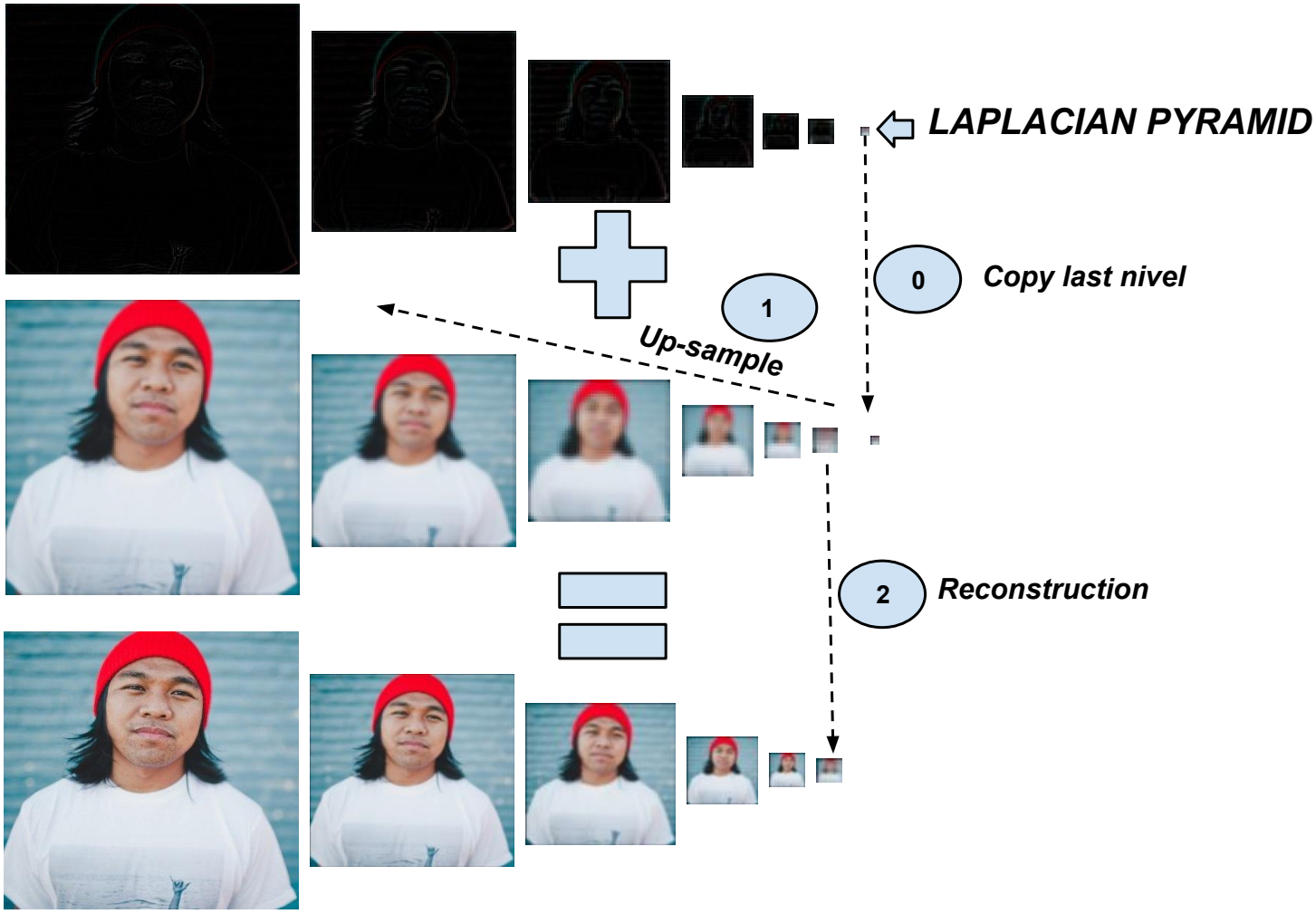


Figure 7: Reconstruction process

```
mid = []
for i in range(level - 1, -1, -1):
    mask = gp_mask.get(i)
    joint = combine(lp_a.get(i), lp_b.get(i), mask)
    mid.append(joint)
```

Finally, the same process of reconstruction as in Laplacian pyramid is done, this yields in the blended image. As explained in the previous section, the function $down$ of the Laplacian pyramid will upsample *img_blend*(the first time *img_blend* is equivalent to the union of the last levels of Gaussian pyramids) and will sum it up with $mid[i]$, this operation is the inverse of the up operation used to build the Laplacian pyramid.

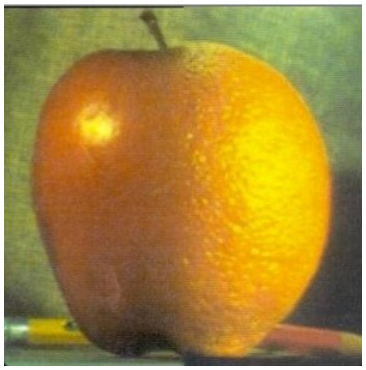
```
img_blend = mid[0]
for i in range(1, len(mid)):
        img_blend = lp_a.down(img_blend, mid[i])
return img_blend
```

The main problem during implementation was related with the size of the up sampled image and the current level: when the width or height was odd, we "lose" a row or a columns. Another problem was related with multi-channel images, it is better to have a function that blends RGB or grayscale images without extra parameters or more complicated stuff, in a transparent way to the user. Thus, our implementation overcomes these problems and has not limitation in this sense, moreover there is a previous step that checks whether the size of the input images are equal such that it resize them to get valid shapes. Note that the implementation allows to set the $mask$ with left-right blending, bottom-up blending and also load a binary image as $mask$.

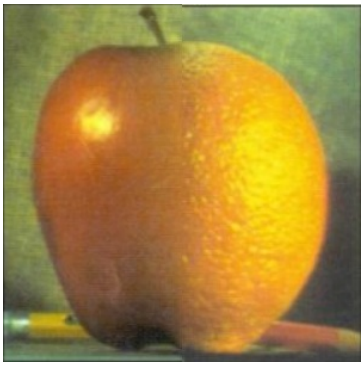
We performed extensive experiments with different images. The results of the blending are shown in the sub figures 8b, 8c and 8d. The blending generated by the OpenCV algorithm is shown in the sub figure 8a. The result of our implementation is very similar to the OpenCV one, therefore, we conclude that our implementation is going well.

6 Exploring Fourier Space

The results for the phase and magnitude extraction are shown in figures 9, 10, 12 and 11.



(a) blending with *OpenCV*.



(b) Left-right blending

(c) Bottom-up blending

(d) Binary image blending

Figure 8: Blending results

Getting the lowest values(figure 9)of the magnitude yields in images that shows groups of points that represents the woman of the original image. Note that the first image (the 1st lowest) seems to not have relation with the original image.

We can see in the figure 10, that the effect of taking the 1st greatest value creates an image that seems to not have

(a) 1st lowest (b) 25 % lowest (c) 50 % lowest (d) 75 % lowest (e) 100 % lowest

Figure 9: Low magnitude extraction

(a) 1st highest (b) 25 % highest (c) 50 % highest (d) 75 % highest (e) 100 % highest

Figure 10: High magnitude extraction

relation with the original one, but, in the rest of images the difference is really small, it is just a little more clear while taking higher values.

(a) 1st lowest (b) 25 % lowest (c) 50 % lowest (d) 75 % lowest (e) 100 % lowest

Figure 11: Low phase extraction

Also, we can see in the figure 11 that while we take less quantity of lower values, the intensity of the image is lower. This occurs because the phase carries a large part of the spatial information of the image, therefore, while we take more values, then major features of the image are preserved.

(a) 1st highest (b) 25 % highest (c) 50 % highest (d) 75 % highest (e) 100 % highest

Figure 12: High phase extraction

Similar to above, in the figure 12 we can see that the intensity of the image is less while taking less quantity of high values. Also we can see that in the sub figure 12c the image suffer a modification, creating a woman of inverted shape next to the original woman. This occurs because we take only high values altering the original spacial characteristics.

Comparing results from images 9 and 11, we can see that the changes in phase have a bigger effect (e.g. we lose a lot of information), moreover, comparing 10 and 12, the results of getting the highest values in the magnitude seems to not have any effect, on the other hand, getting the 75 % of the highest phase values yields in big changes. Thus the phase is more important than the magnitude.

7 Frequency Blending

In order to blend the images using the frequency domain, we convert the masked image to the frequency domain and focus on approaches to combine these frequency domains images. We consider 6 ideas of combination.

- The first one is named "left-right", this method takes a left half of the first frequency domain image and the right half of the another. The results of this method are shown in figure 13. These images shows a successful combination of the original images but the intersection point is not as smooth as it would be expected, this is because in the frequency domain it is not possible to smooth just a small part of the image (e.g. intersection points), the alternative of blurring (e.g. taking out low frequencies) before applying the left-right blend would end up in the same result but blurred in the whole image and not just the intersection points.

(a) Left-right blending apple-orange (b) Left-right blending woman-man

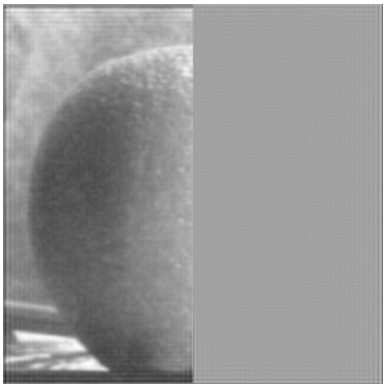
Figure 13: Left-right blending

- The second method is named "bottom-up", this is similar to the "left_right" approach but takes the up half of one image and the bottom half of the another. The results of this method are shown in figure 14. As equal to the previous approach, the images were successfully joined but with a sharp change in the intersection. Different from previous approach, some vertical shadows appear (left-right approach has horizontal shadows), this highlights the intersection change.

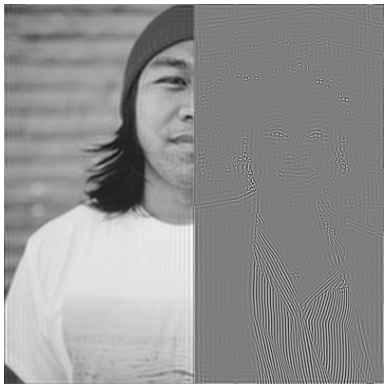
(a) Bottom-up blending apple-orange (b) Bottom-up blending woman-man

Figure 14: Bottom-up blending

- The third method is named "centered", this takes the lowest frequencies from one image and the highest frequencies from the other. The results of this method are shown in figure 15. Different from previous methods, this approach can not joint the images correctly, it generates a strange combination, this happens because we are taking the lowest frequencies of the second image. Note that in the second combination, the low frequency part is more visible, this due to the quantities of low frequencies that woman image contains.
- The fourth method is named 'chessboard', it combines the frequency images taking one pixel from one image and the next from the other, this creates a kind of chessboard of the frequency images. The results of this method are shown in figure 16. Due the extreme combination of frequency images, the characteristics of the images mix up generating shadows and photographic negatives.
- The fifth method is named "sliding-rows", this creates horizontal stripes intercalating values from the both frequency images. The results of this method are shown in figure 17. These results are interesting because the same method gives really different results, in the case of the apple-orange blending the left side of the image

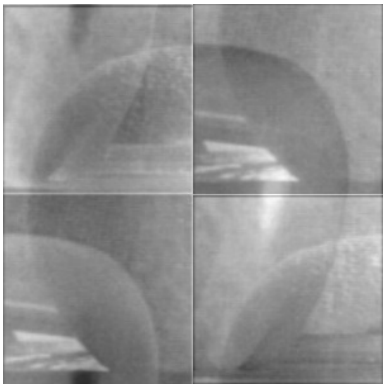


(a) Centered blending apple-orange



(b) Centered blending woman-man

Figure 15: Centered blending

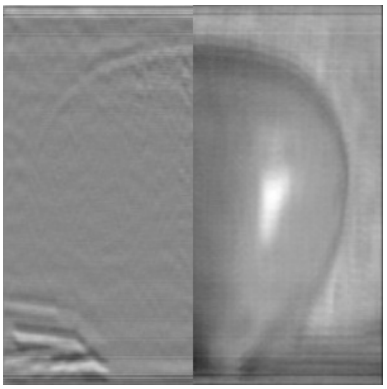


(a) Chessboard blending apple-orange

(b) Chessboard blending woman-man

Figure 16: Chessboard blending

look like a kind of texture of the image (something like a local binary pattern), and the woman-man blending is much more better, the explanation for this effect may be related with the properties of the images, as equal as is centered blending.



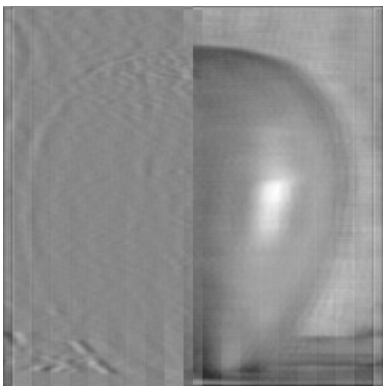
(a) Sliding-rows blending apple-orange

(b) Sliding-rows blending woman-man

Figure 17: Sliding-rows blending

- The sixth method is named "sliding-columns", this creates vertical stripes intercalating values from the both frequency images. The results of this method are shown in figure 18. The experiments looks similar to the previous approach but in this case the images are a little bit more lighter.

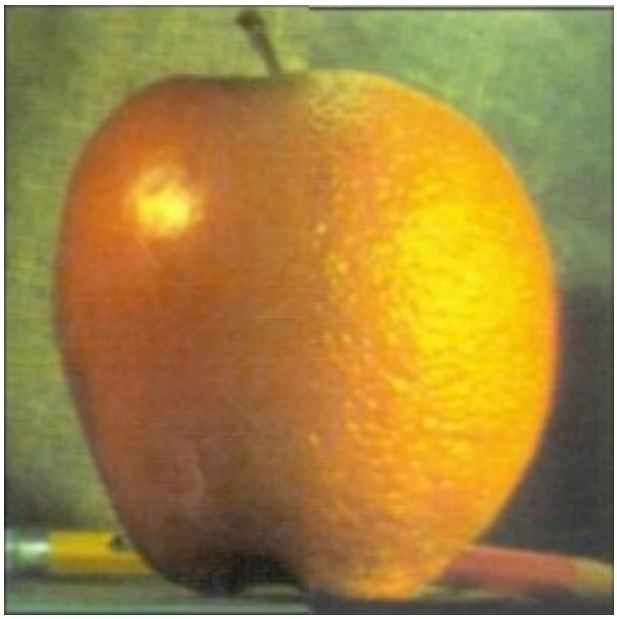
Comparing this results with the obtained in the space domain (figure 19) we can see that the change in the space domain is smoother, this is because the blending in the frequency domain is not fully development. Note that the frequency domain blending only considers grayscale image as inputs.



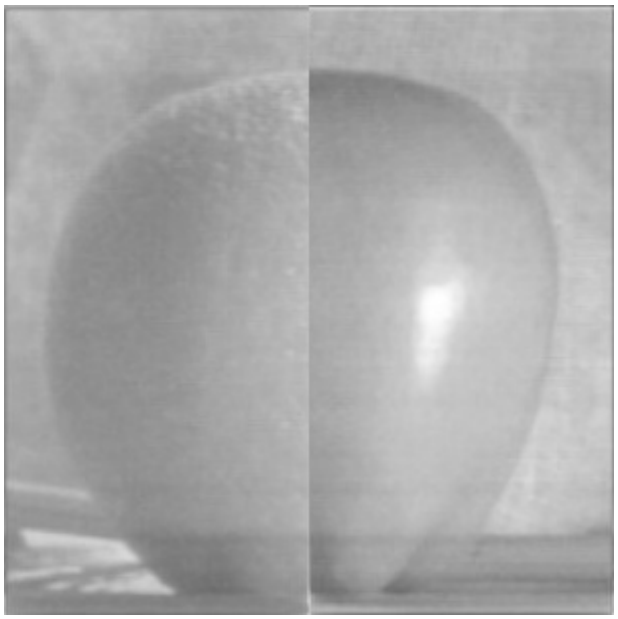
(a) Sliding-columns blending apple-orange

(b) Sliding-columns blending woman-man

Figure 18: Sliding-columns blending



(a) Space domain blending



(b) Frequency domain blending

Figure 19: Blending comparison