Homework 1

Thiago Henrique Pincinato
Introduction to Signal and Image Processing

March 17, 2019

1 Regular Tesselation

1.1

The three shapes that satisfy the two conditions of Regular Tesselation are :

- Equilateral Triangle

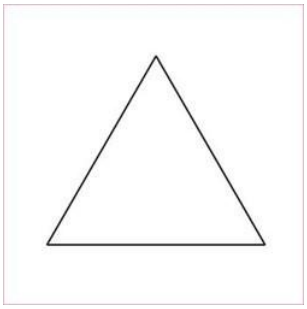

Figure 1: *Triangle*

- Square

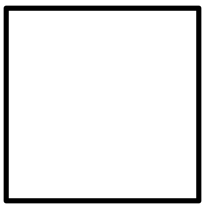

Figure 2: *Square*

- Regular Hexagon

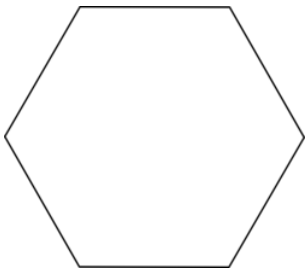

Figure 3: *RegularHexagon*

1.2

Proof. First, we need to remember the equation :

$$Sum\_of\_all\_angle = 180(\alpha - 2)$$

Where α is the number of polygon's angles.

Second, each angle of a polygon can be computed as:

$$angle = \frac{Sum\_of\_all\_angle}{\alpha}$$

Third, by using the two equations above, we can formulate that :

$$\frac{(180(\alpha - 2)e)}{\alpha} = 360 => \frac{(\alpha - 2)e}{\alpha} = 2$$

Where e is the number of elements needed to cover one of the vertices. In the figure 4, we can identify that $e = 3$

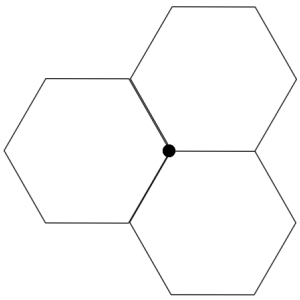

Figure 4: *number e of a Regular Hexagon*

Finally, by mathematical manipulation, we have:

$$\alpha e - 2e = 2\alpha => e = \frac{2\alpha}{(\alpha - 2)}$$

This equation must fulfill certain criteria, such as , e and α should be an integer and positive number. That leads to the following conditions:

- $\alpha >= 3$
- In exception to the number 3, α must be a odd number . Otherwise, e would be a fraction.
- 2α must be a divisible by $\alpha - 2$.

By following those conditions, one can realize that the only possible α numbers are 3 (triangle), 4 (square) and 6 (hexagon). □

2 Lloyd-Max quantization

2.1

In order to minimize δ with respect to z_k, the derivation of δ with respect to z_k should be zero, and the second derivation should be bigger than zero.

$$\delta = \sum_{k=1}^{K} \int_{k}^{k+1} (z - q_k)^2 p(z) dz => \frac{\partial \delta}{\partial z} = \frac{\partial \int_{k-1}^{k} (z - q_k)^2 p(z) dz + \int_{k}^{k+1} (z - q_k)^2 p(z) dz}{\partial z}$$

$$=> (z - q_{k-1})^2 p(z) - (z - q_k)^2 p(z) => (q_{k-1} - q_k)(q_{k-1} + q_k - 2z) p(z) = 0$$

By assumption $p(z) \neq 0$ *and* $q_{k1} \neq q_k$, which leads to :

$$(q_{k-1} + q_k - 2z) = 0 => z = \frac{q_{k-1} + q_k}{2}$$

2.2

In order to minimize δ with respect to q_k, the derivation of δ with respect to q_k should be zero, and the second derivation should be bigger than zero.

$$\delta = \sum_{k=1}^{K} \int_{k}^{k+1} (z - q_k)^2 p(z) dz => \frac{\partial \delta}{\partial q_k} = \frac{\partial \sum_{k=1}^{K} \int_{k-1}^{k} (z - q_k)^2 p(z) dz}{\partial q_k}$$

$$=> \frac{\sum_{k=1}^{K} \partial \int_{k}^{k+1} (z - q_k)^2 p(z) dz}{\partial q_k} => \sum_{k=1}^{K} \int_{k}^{k+1} 2(z - q_k) p(z) dz$$

$$=> \sum_{k=1}^{K} \int_{k}^{k+1} 2(-q_k) p(z) dz + \sum_{k=1}^{K} \int_{k}^{k+1} 2(z) p(z) dz = 0 => (-q_k) \int_{k}^{k+1} p(z) dz = - \int_{k}^{k+1} (z) p(z) dz$$

Finally;

$$q_k = \frac{\int_{k-1}^{k} (z) p(z) dz}{\int_{k-1}^{k} p(z) dz}$$

2.3

The Lloyd−Maxwell method can be interpreted as a minimization problem, in this case minimization of δ. One could analyze such a method as coordinate descent, in which the current iteration determines the next step and direction in the other coordinate.

On those situations, it could happen that an displacement in qk would lead to a displacement in zk in the opposite direction of its minimum. In the next iteration, zk would search for its minimum again, moving qk to the previous position. In a next step, qk would force zk to dislocate in opposite of its minimum again, and therefore, the algorithm would never converge.

Besides those situations and excluding approximation issues, the algorithm would converge, leading to optimize quantization intervals. Those intervals would have different sizes, but approximately equal pixel population.

3 Chamfer distances

The Chamfer algorithm was implemented by following the instructions given in the exercise. The results can be seen in the figure 5.

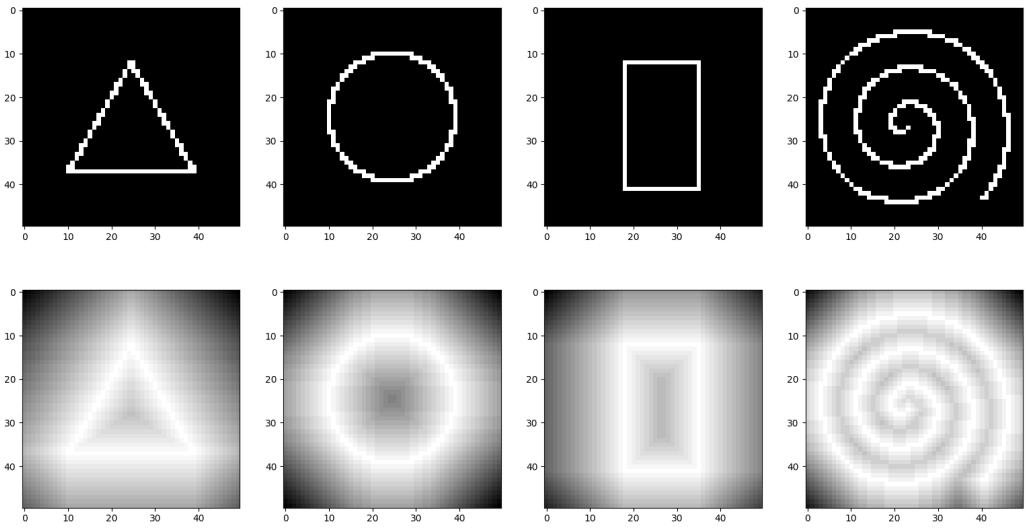

Figure 5: *Result of the Chamfer algorithm*

In order to mitigate the boundary effects generates by the Chamfer algorithm, a new step (step 6) was implemented. The new step consists of returning in the missing distances (pixels), applying a new kernel that fit the boundaries and computing :

$$distance\_map(x) = min[L1(x, q) + distance_map(q)]$$

The result represents exactly what was expected, a distance map of the shape images. This map cover the direction of a well defined object (shaped object), and thus, it has been used for shape-based object detection.

4 Bilinear interpolation

Bilinear interpolation is a mathematical operation in an 2D object. The main idea is the realization of a 1D interpolation in x for each new y position.

Despite its name, a Bilinear interpolation is rather quadratic than linear. In addition, this tecnhique is used for re-scaling images. When doing so, the bilinear interpolation avoid holes (pixels that are not assigned proper values). Those pixels values are computed by taking into account the value of the neighbors pixels.

Bilinear interpoaltion allows an approximation of intensities value for those holes.

4.1

A linear interpolation was implemented by using the formula:

$$y = y1 * \frac{x - x0}{x1 - x0} + y0 * (1 - \frac{x - x0}{x1 - x0})$$

The code can be seen in the figure 6.

```
y_new = np.zeros(x_new.shape[0])
x0 = np.concatenate(([0], x_vals), axis=0);
x1 = np.roll(x0,-1)
y0 = np.concatenate(([0], y_vals), axis=0);
y1 = np.roll(y0,-1)
for i in range(0, x_new.shape[0]):
    if x_new[i] >= np.max(x0):
        y_new[i] = y0[y0.shape[0] - 1]
    else:
        index = np.min(np.where(x0 > x_new[i])) -1
        if index == 0:
            y_new[i] = y0[1]
        else:
            y_new[i] = y1[index]*((x_new[i] - x0[index])/(x1[index] - x0[index]))
            y_new[i] = y_new[i] + y0[index]*(1 - (x_new[i] - x0[index])/(x1[index] - x0[index]))
return y_new
```

Figure 6: *Linear_interpolation*

The code was tested with the vector x = [1, 2, 3, 4, 5, 6, 7, 8] , y = [0.2, 0.4, 0.6, 0.4, 0.6, 0.8, 1.0, 1.1] and new_x = [0.5, 2.3, 3, 5.45], generating the output = [0.2, 0.46, 0.6, 0.69].

4.2

The 1d interpolation can be ssen in the figure 7:

As previously, the code was verified with a defined vector and a known output. The results show a 100 percent matching.

```
x = np.arange(0, signal.shape[0], 1)
xnew = np.linspace(0,signal.shape[0]-1, signal.shape[0]*scale_factor)
signal_interp = interp(signal, x, xnew)
return signal_interp
```

Figure 7: $1D\_Linear\_interpolation$

4.3

4.3.1

The result of the re-scaling gray image using interpolation can bee seen in the figure 8 and figure 9

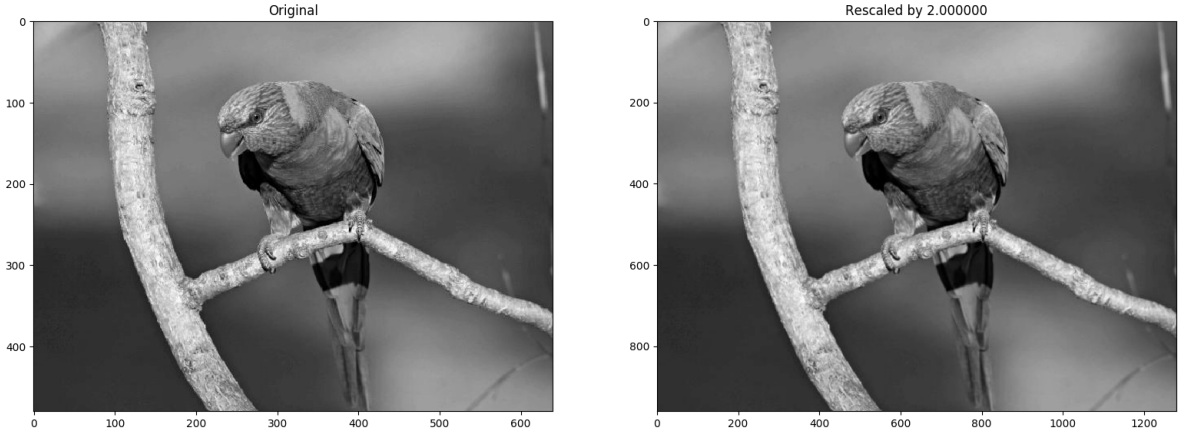

Figure 8: $2D\_Linear\_interpolation$

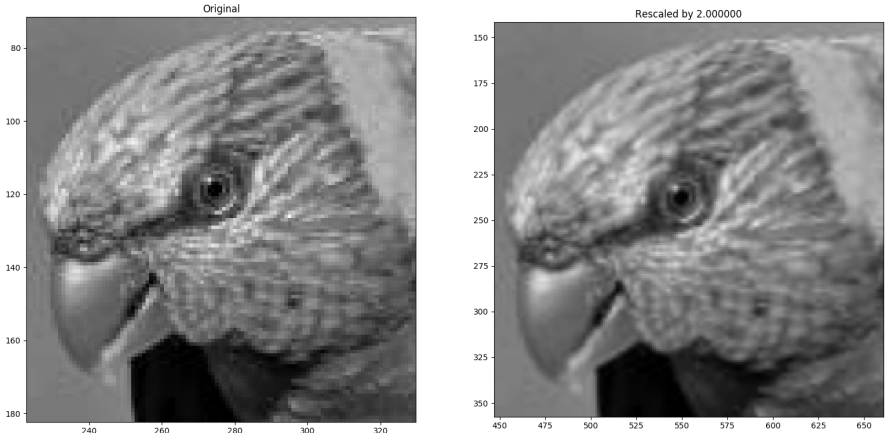

Figure 9: $birds\_Zoom$

In those images, we can see the smooth/blur effect of the re-scaling by interpolation. As the holes due to the re-scaling are filled by a linear interpolation, its pixel value is approximately the average between the neighbor pixel in x and y directions. Therefore, the re-scaled image is slight blurred

4.3.2

In a RGB image, the bilinear interpolation is applied on each channel separately. Besides that, the processes and results are exactly the same as in gray images. The images 10 and 11 prove such assumption.

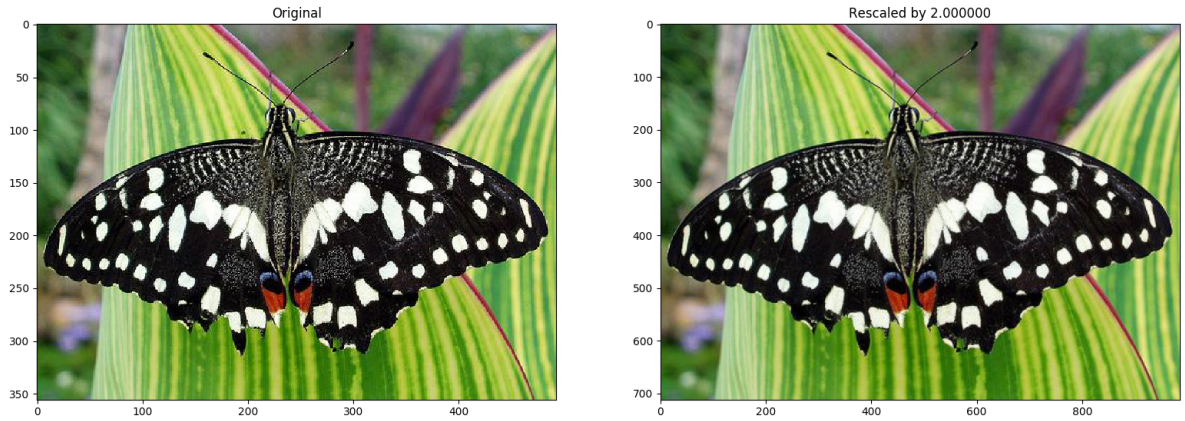

Figure 10: *butterfly*

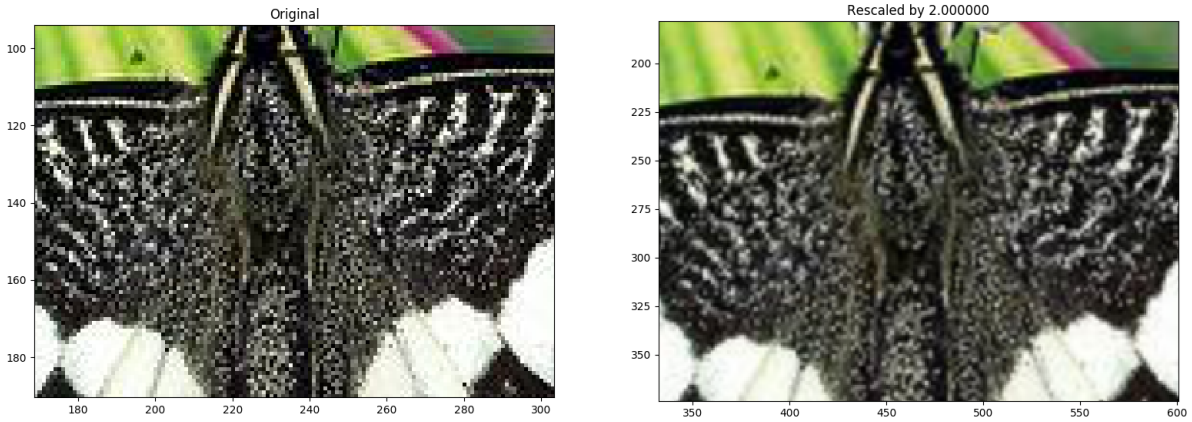

Figure 11: *butterfly_Zoom*