Project 4

Isabela Telles Furtado Doswaldo - RA: 170012

MC920 - Introduction of Digital Image Processing
University of Campinas

1 Introduction

The goal of the project is to implement algorithms to perform geometric transformations of image scaling and image rotation. Both are often necessary in order to align images that are taken at different times, or with different sensors, as well as to correct effects of camera orientations.

A geometric transformation can be accomplished in two steps: spatial transformation, which consists in reorganize the pixels of the image mapping each point of the image to another point in a new coordinate system, and intensity interpolation, which consists in the attribution of intensity levels to the pixels of the transformed image.

The program developed in this project implements both, image scaling and rotation, using Python, along with the following libraries: OpenCV, Numpy and Math.

The images used for testing are available in http://www.ic.unicamp.br/~helio/imagens_png/.

2 Methodology

The program was designed using four different methods of interpolation, so the user can choose which one is going to be used to scale or to rotate the image. Thus, it requires the method as an argument, but it also requires more arguments, as you can see below:

1. Input image: path to the PNG image file before the alignment
2. Output image: path to the PNG image file after the alignment
3. Interpolation method: an integer between 0 and 3
   (a) 0: Nearest neighbor interpolation
   (b) 1: Bilinear interpolation
   (c) 2: Bicubic interpolation
   (d) 3: Lagranges' interpolation
4. Angle of rotation (-a): measured in degrees counterclockwise and it can be a float number
5. Scale factor (-s): it can be a float number
6. Output image dimension (-d): two integers, the first being the height of the transformed image, and the second being the width

It is important to point out that the program performs one of the geometric transformations (rotation or scaling), never both.

See examples of usage to execute the program:

```
python3.5 transform.py baboon.png transformed_baboon.png 0 -a 48.75
```

It rotates the baboon.png in 48.75° using nearest neighbor interpolation.

```
python3.5 transform.py baboon.png transformed_baboon.png 2 -s 1.75
```

It scales the baboon.png in a factor of 2, resulting in an image with height of 1024 and width of 1024, using bicubic interpolation.

```
python3.5 transform.py baboon.png transformed_baboon.png 3 -d 750 980
```

It scales the baboon.png in an image with height of 512 and width of 1024 using Lagrange's interpolation.

After the program initiates, the image is loaded from a file into a 3D numpy array using the OpenCV library's function *cv2.imread()* along with the flag `cv2.IMREAD_GRAYSCALE`, which allows the image to be read in grayscale mode.

2.1 Image Scaling

As seen in examples of usage, you can either provide the scale factor or the dimension of the output image in order to perform an image scaling. Regardless of which one you choose to provide, the program will calculate the scale factor of axis y (s_y) and axis x (s_x).

The scale factor can be a float number, so first of all it is rounded to 3 digits after the decimal point, which will give a more accurate precision for the calculation of the new width and height of the image. Each axis has its scale factor equaled to the rounding scale factor.

The (height, width) dimension of the output image is divided by the height and width of input image, respectively, in order to calculate s_y and s_x. Then, it is rounded to 3 digits after the decimal point, for the same reasons of rounding the provided scale factor.

The new height and width are calculated by multiplying the scale factor y and x for the height and width of the input image, and then rounding it to an integer number.

In order to map the points of the scaled image (x_o, y_o) to the points of the input image (x_i, y_i), each index x of columns and y of rows of the new image will be divided by the scale factor of its axis, as showed in equation (1) and (2).

$$x_o = x_i / s_x \tag{1}$$

$$y_o = y_i / s_y \tag{2}$$

To improve the performance of the program, the float values of rows' and columns' map indexes will be stored in numpy arrays X and Y, each one storing all of the x_o and y_o calculated.

Since every x_o and y_o are float numbers, the coordinates (x_o, y_o) may not be integers. The interpolation methods are used to approximate the coordinates, then the values of pixels can be successfully attributed.

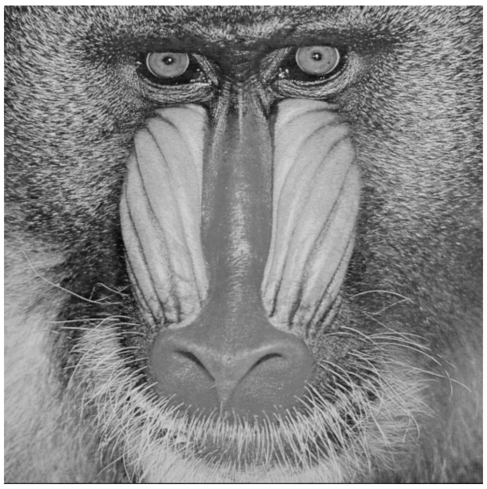

Figure 1: Input Image

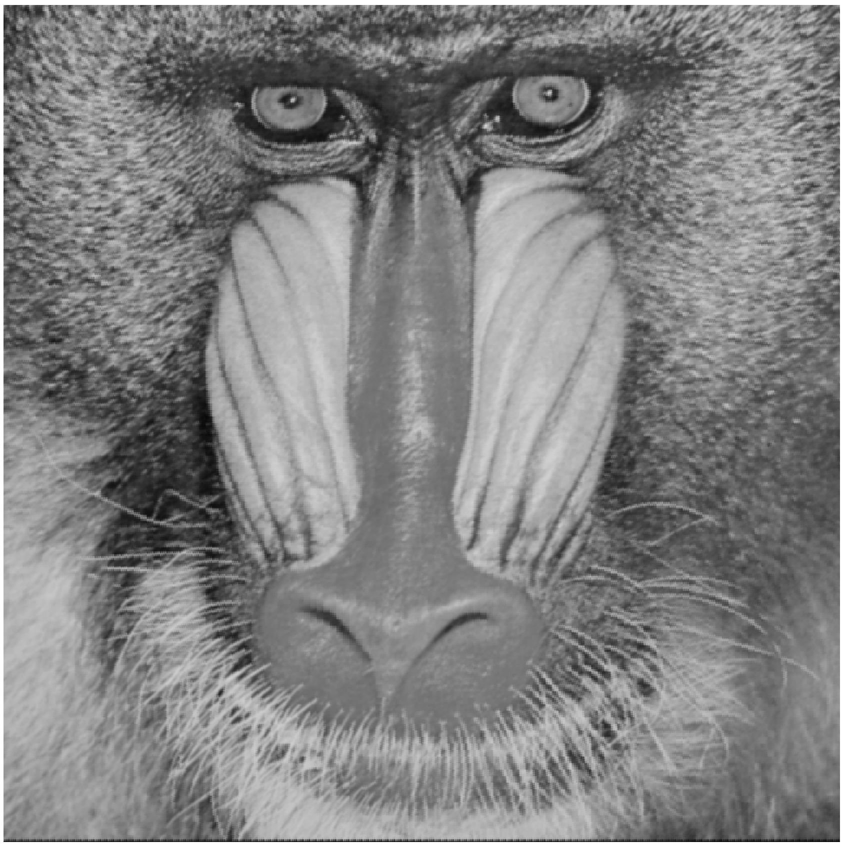
Figure 2: Image scaled in 1.75 using bicubic interpolation

2.2 Image Rotation

In order to map the points of the rotated image to the points of the input image, the program uses the widely known transformation matrix seen below:

$$T = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

The equation (3) and (4) show how a rotation can be performed around the origin if the image, top left corner, given a numpy array X_i of columns' indexes, with shape (1, width), and a numpy array Y_i of rows' indexes, with shape (height, 1). The results are numpy arrays X and Y of 2 dimension, representing the coordinates of the input image in which the output image will be mapped.

$$X = X_i \cos\theta - Y_i \sin\theta \tag{3}$$

$$Y = X_i \sin\theta + Y_i \cos\theta \tag{4}$$

The rotation of the image around the origin may not provide the best result, since in a rotation of 90^o counterclockwise, for instance, the image won't fit in the current dimension of the imagem. That's why the program performs a rotation around the center of the image, using both equations (5) and (6), with c_x and c_y representing the center of axis x and y, obtained dividing the height and width of the input image by 2.

$$X = (X_i - c_x)\cos\theta - (Y_i - c_y)\sin\theta + c_x \tag{5}$$

$$Y = (X_i - c_x)\sin\theta + (Y_i - c_y)\cos\theta + c_y \tag{6}$$

Since the numpy arrays X and Y can contain indexes out of the limits, the program limit those indexes to zero after it creates a new image from the input image with white padding of thickness

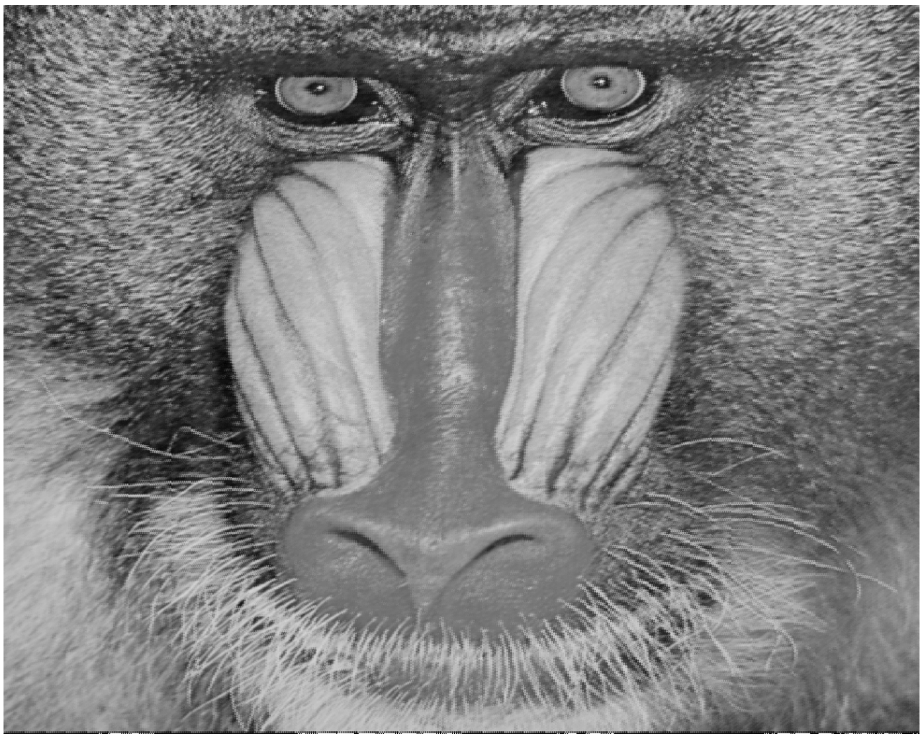
Figure 3: Image scaled in 780x980 using Lagranges' interpolation

one. Then, the indexes out of the limits will be mapped to the point (0, 0), which will have a intensity value of 0 because of the padding and the image new background will be white.

Even performing the rotation around the center, the image is still cropped, because the corners don't fit in the original height and width of the image.

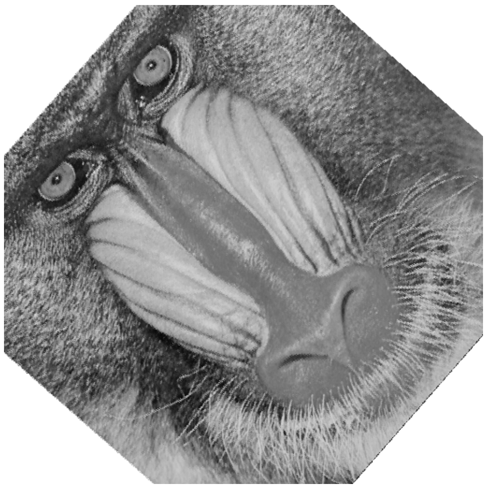
Figure 4: Image rotated in 48.75° counterclockwise using Lagranges' interpolation

2.3 Nearest Neighbor Interpolation

The nearest neighbor interpolation is one of the easiest interpolation to implement. The numpy array of float indexes is simply rounded half up. Numpy function *np.where()* verifies if the element of the array has its fraction less than 0.5, if so the element is switched to its floor value, otherwise the element is switched to its ceil value.

The intensity of levels are attributed to pixels of the new transformed image through indexing. The image scaling provides numpy arrays of 1 dimension, so the attribution involves slicing, as you can see in (7).

$$new\_image = image[X,:][:,Y] \tag{7}$$

The image rotation, on the other hand, provides numpy arrays of the input image dimension, so the attribution consists in what you can see in (8).

This type of interpolation is the fastest in comparison with the three others implemented by the program, but there are several problems related to it, as the rough edges of the output image which can be seen carefully in Figure 5 and 6.

$$new\_image = image[X,Y] \tag{8}$$

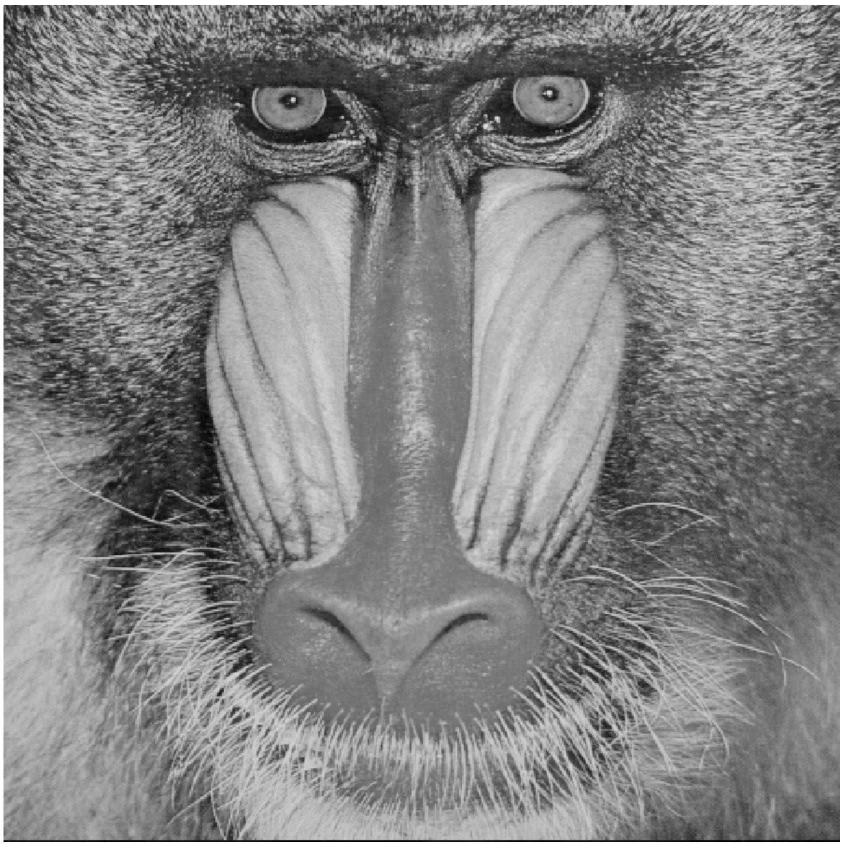

Figure 5: Image scaled in 1.75 using nearest neighbor interpolation

2.4 Bilinear Interpolation

The bilinear interpolation uses a weighted average distance of the nearest four neighbors to determine the intensity of each pixel in the transformed image. It is given by the equation (9), which uses dx and dy, both given by equations (10) and (11), representing the fractional part of rows' and columns' indexes.

$$\begin{gathered} f(x_o,y_o) = (1-dx)(1-dy)f(x,y) + dx(1-dy)f(x+1,y)+ \\ (1-dx)dyf(x,y+1) + dxdyf(x+1,y+1) \end{gathered} \tag{9}$$

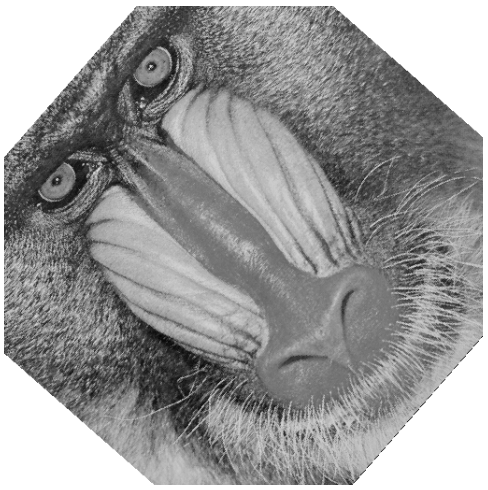
Figure 6: Image rotated in 48.75° counterclockwise using nearest neighbor interpolation

$$dx = x_o - x = x_o - \lfloor x_o \rfloor \tag{10}$$

$$dy = y_o - y = y_o - \lfloor y_o \rfloor \tag{11}$$

In comparison with the technique of nearest neighbor interpolation, the bilinear interpolation is better in terms of quality of the image, but the problem of the edges persists, as you can see in Figure 7 and 8.

2.5 Bicubic Interpolation

The bicubic interpolation uses a neighborhood of 4 x 4 points around each point to calculate its value of intensity. It is given by the equation (12), which also uses dx and dy, both given by equations (10) and (11).

$$f(x_o, y_o) = \sum_{m=-1}^{2} \sum_{n=-1}^{2} f(x+m, y+n) R(m-dx) R(dy-n) \tag{12}$$

$$R(s) = 1/6[P(s+2)^3 4P(s+1)^3 + 6P(s)^3 4P(s1)^3] \tag{13}$$

$$P(t) = \begin{cases} t & t > 0 \\ 0 & t \leq 0 \end{cases} \tag{14}$$

This interpolation reduces the rough edges of the image, which can be seen in Figure 2 and 9, but it is the most costly algorithm of all performed by the program.

2.6 Lagranges' Interpolation

The Lagranges' interpolation uses a neighborhood of 4 x 4 points around each point to calculate its value of intensity. It is given by the equation (15), which also uses dx and dy, both given by equations (10) and (11).

$$\begin{aligned} f(x_o, y_o) = &\frac{-dy(dy-1)(dy-2)L(1)}{6} + \frac{(dy+1)(dy1)(dy2)L(2)}{2} + \\ &\frac{-dydy(dy+1)(dy-2)L(3)}{2} + \frac{dy(dy+1)(dy-1)L(4)}{6} \end{aligned} \tag{15}$$

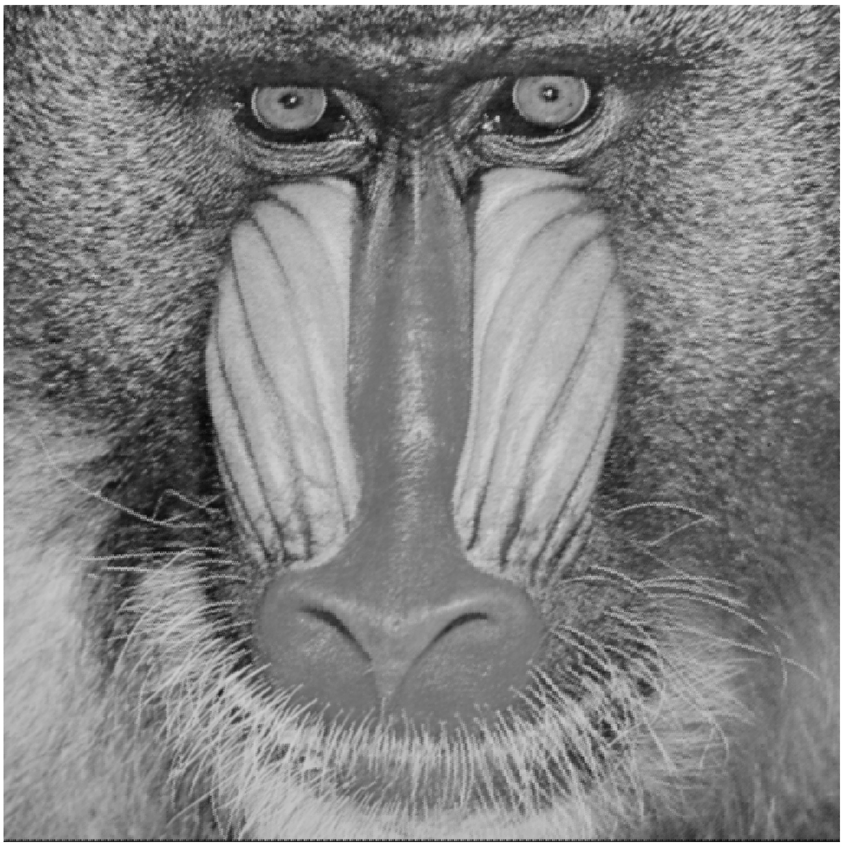

Figure 7: Image scaled in 1.75 using bilinear interpolation

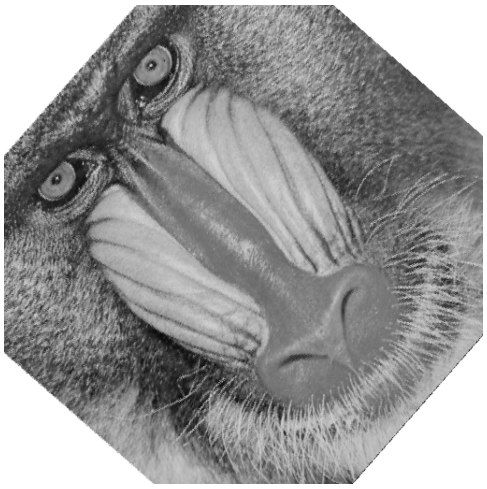

Figure 8: Image rotated in 48.75° counterclockwise using bilinear interpolation

$$\begin{aligned} L(n) = \frac{-dx(dx-1)(dx-2)f(x-1,y+n-2)}{6} + \frac{(dx+1)(dx1)(dx2)f(x,y+n-2)}{2} + \\ \frac{-dxdx(dx+1)(dx-2)f(x+1,y+n-2)}{2} + \frac{dx(dx+1)(dx-1)f(x+2,y+n-2)}{6} \end{aligned} \tag{16}$$

This interpolation has a better performance than bilinear and bicubic, the rough edges are minimized, and it is not available in libraries as OpenCV or Scikit Image. The result examples of geometric transformations using Lagranges' interpolation can be seen in Figure 4 and 10.

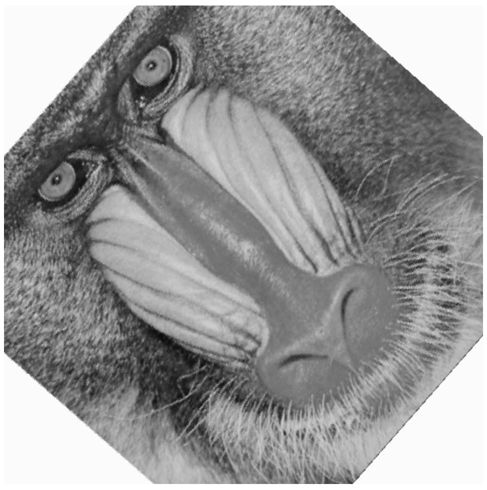

Figure 9: Image rotated in 48.75° counterclockwise using bicubic interpolation

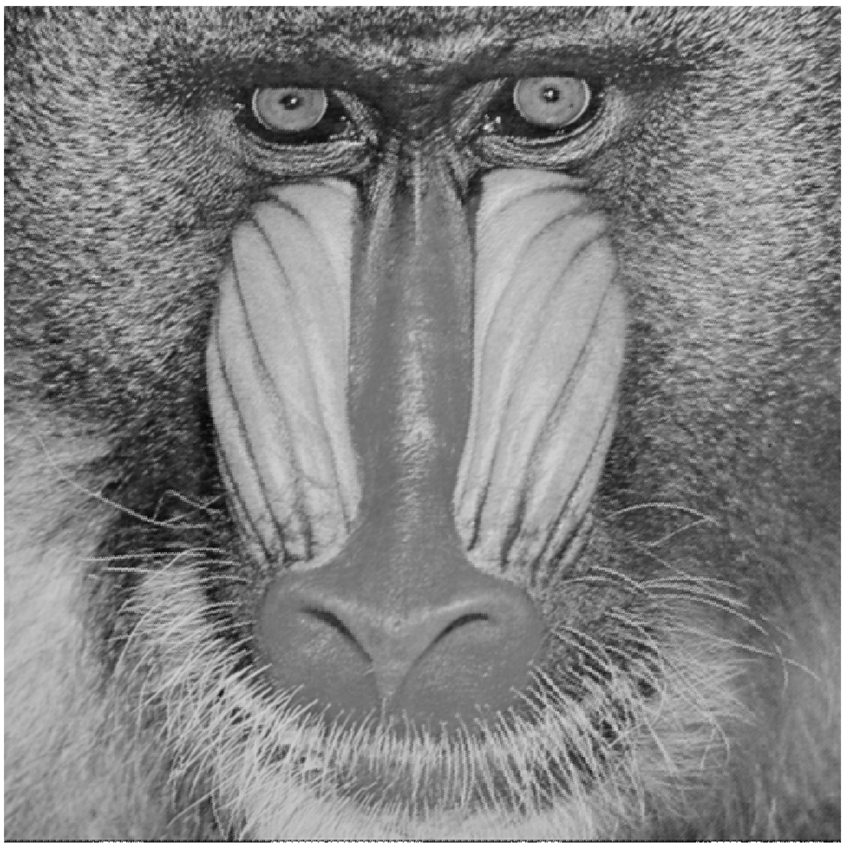

Figure 10: Image scaled in 1.75 using Lagranges' interpolation