Project Report

I. Index

II. Objective

To compare nearest neighbor, bilinear and bicubic interpolations on different Image Geometric Transformation (or to be specific Affine Transformations).

III. Overview

An affine transformation is an important class of linear 2-D geometric transformation. An affine transformation is a function between affine spaces which preserves parallel relations i.e. ratios of distance between points, parallel lines etc. Affine transformations we dealt with in this project are reflection, translation, rotation, shear and scaling. And the interpolation techniques we explored in this project are nearest neighbor, bilinear and bicubic.

IV. Team Assignments

Timothy	–	GUI and Rotation
Matthew	–	Translation, Shear and Scaling
Qaem	–	Reflection, Nearest Neighbor and Bilinear Interpolation
Navneet	–	Bicubic Interpolation and Report

V. Algorithms

i. Transformation

a. Rotation

Image rotation was implemented by finding the centroid of the image and rotating the 4 corners of the input image to get the extent of the rotated image. Next, an empty rotated image was constructed. Then, for each pixel in the empty rotated image, we rotated back to the original image pixel grid to find the unknown pixel locations on input image and each unknown intensity was calculated using interpolation. Initially, image rotation ran into problem because it seems more difficult to do interpolation based in the rotated grid since the known location in the rotated grid might not fall on integer values of row and column.

b. Reflection

Reflection has a simple implementation. For both horizontal and vertical reflections, we only operate on one half of the image, reducing operation time. For horizontal reflection, we only operate on 1st half of rows. We simply swap the top and bottom (with loop moving inwards) pixel for the current column.

Similarly, for vertical reflection, we swap left and right pixels instead of top and bottom (again with loop moving inwards).

c. Translation

We determine the left, right, top, and bottom padding based off the desired x and y translation values. Positive values move the image down and towards right, negative values shift the image up and towards left. 'np.pad' function is then used to add 0s as needed based on the padding values (black pixels indicating the shift in the appropriate direction). Then, for each pixel we take the pixel intensity and move them to the location padding + original_location. The only difficult part was remembering to swap the left/right and up/down order in the image. Since x is shifting the column value, it comes second. Y is shifting the rows up or down, so it comes first.

d. Scaling

Scaling was implemented by taking the values for new dimensions i.e. desired width and height We then calculate scaling factors 'fx' and 'fy' values based on the input image dimensions and the desired dimensions. Next, we create an empty image (i.e. filled with zeroes) based on new image dimensions. For each pixel we find 4 neighboring pixels and calculate interpolated intensity for the current position using appropriate interpolation method.

e. Shearing

The shearing algorithm ended up being straightforward, once we got the hang of it. To calculate the new image width, we take the original width plus the height multiplied by the absolute value of the multiplier 'm'. The absolute value is important because negative values of 'm' will result in a wider image. For each pixel in the new image, we calculate the value 'x' as 'i – m*j'. If this value is outside the bounds of the original image, we set the value of the new pixel to 0 (black). Otherwise, the intensity is calculated using appropriate interpolation method.

By default, this method works for positive values of 'm' and only in the vertical direction. Rather than trying to make the algorithm complex, we transformed the image matrix into one which matches the setup. For negative 'm', the issue is that the image shears from the origin. Since for negative 'm' pixels extend back past the origin, this results in pixels being cut off, as if it had been translated. To counter this, we flip the image horizontally and set 'm' as positive if 'm' is negative. We flip it back again, after the new image is calculated.

For horizontal shearing, we could implement a different algorithm which has essentially the same formulas but targeting different values, i.e. y maps to 'y = j – m*i'. However, we can reduce it to the previously solved problem by rotating the matrix before and after. When the matrix is rotated 90 degrees, it matches the matrix which can be sheared vertically. Then once the new image is calculated, rotate it -90 degrees (i.e. back to the original orientation). In both cases, we only need to undo them in the opposite order we did them in. To summarize, our method: 'rotate -> flip -> calculate-> flip -> rotate'.

ii. <u>Interpolations</u>

a. Nearest Neighbor

Nearest neighbor is the simplest and fastest implementation of image scaling technique. This technique finds the nearest neighbor with known intensity value and uses the same intensity for current unknown pixel. It is very useful when speed is the main concern e.g. when zooming thumbnail for preview.

b. Bilinear

Bilinear is like nearest neighbor except the fact that we interpolate the intensity of current pixel depending upon neighboring pixels (4 neighbors)

c. Bicubic

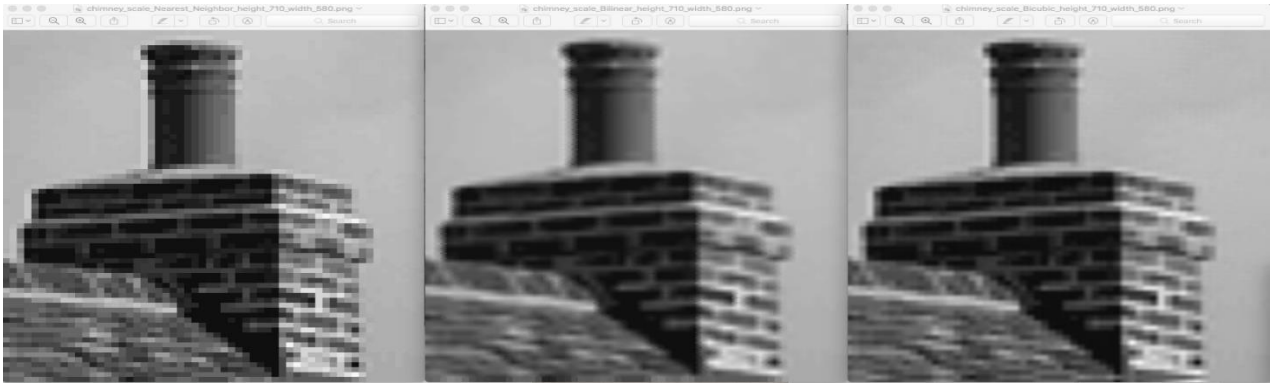
Bicubic interpolation is again like bilinear except the way we perform interpolation. Similar to bilinear, we find 4 surrounding input image pixels and then find x-derivative, y-derivative and cross-derivative for each neighboring pixel. These 16 coefficients are then used in the formula described below to get the interpolated value. 'w' and 'h' are distance of unknown pixel from neighboring right and bottom pixels respectively. P is the interpolated value.

$$p(J,K) = \sum_{m=0}^{3} \sum_{n=0}^{3} a_{mn}(1-w)^m(1-h)^n$$

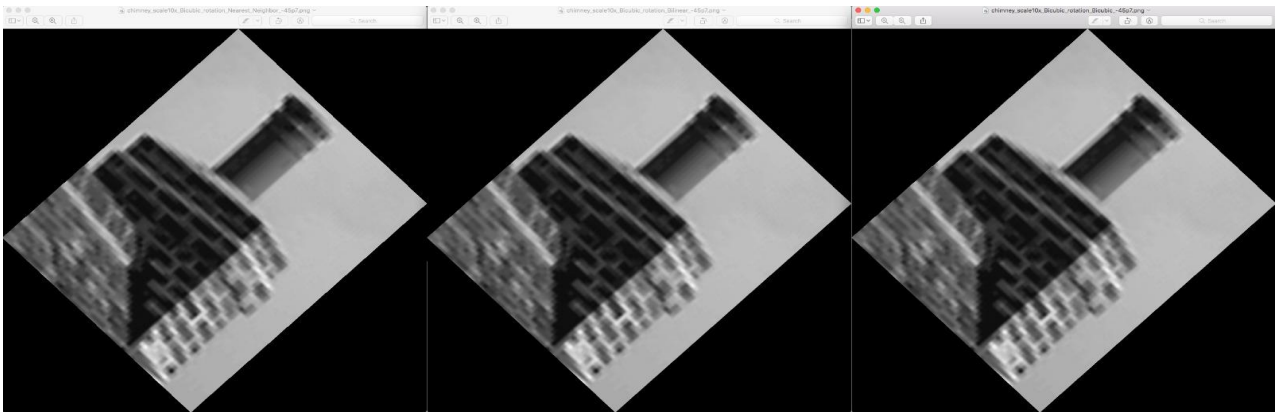
VI. Results

i. Interpolation Comparisons (Nearest Neighbor vs Bilinear vs Bicubic)

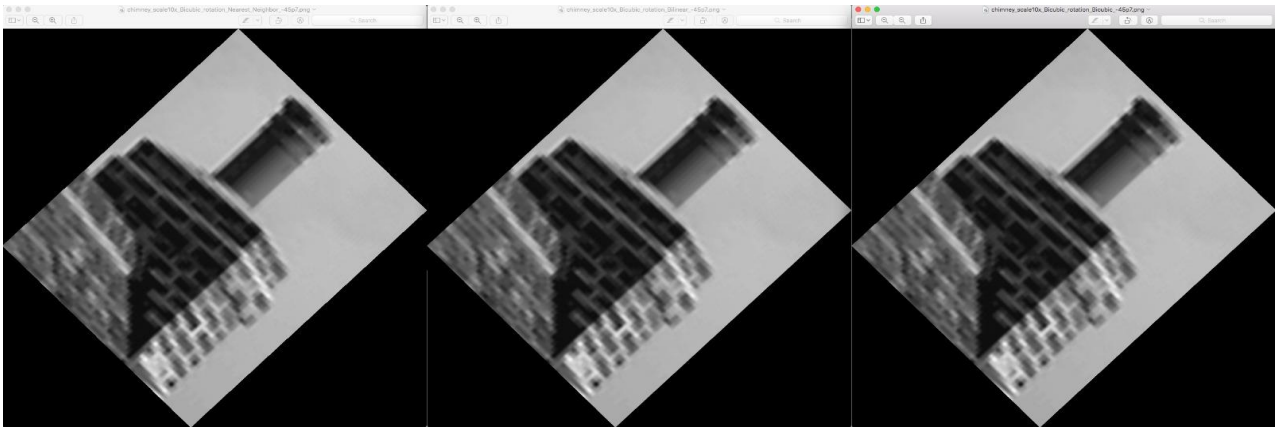
a. Scaling



b. Rotation



c. Shear



ii. Translation & Reflection

VII. Operation Time Statistics

Interpolation → / Operation ↓	Nearest Neighbor (in Seconds)	Bilinear (in Seconds)	Bicubic (in Seconds)
i. Rotation (58 * 71) (tested with multiple angles)	0.77 – 2.03	1.42 - 2.45	
ii. Scaling (500 * 500) Scaled to 250*250	0.08 - 0.12	2.25 - 2.76	
iii. Shearing			

VIII. GUI

We used 'tkinter' for the GUI for our project. Right now, it configures all images to be displayed as 400 x 400 pixels in the GUI display. This means that non-square images should be distorted.

IX. Conclusion

In this project, we successfully explored three interpolation methods i.e. nearest neighbor, bilinear and bicubic for performing affine transformations. We covered reflection, translation, rotation, shear and scaling as a part of geometric image transformation. We discovered that out of all the transformations we performed, result of scaling, rotation and shear is dependent on interpolation technique used. Bicubic and bilinear technique produced almost similar results. They both were much better when compared with nearest neighbor algorithm.

X. Future Works

i. Rotation algorithm efficiency can be improved to reduce operation time.
ii. Reflection for the case x=y (i.e. along diagonals) can be added.

XI. References

i. https://en.wikipedia.org
ii. http://tech-algorithm.com