IMAGE GEOMETRIC TRANSFORMATION

Team: MMFSN

Members: Michael Tran, Matthew Lai, He Feng, Raga Shalini Koka, Navya Doddapaneni

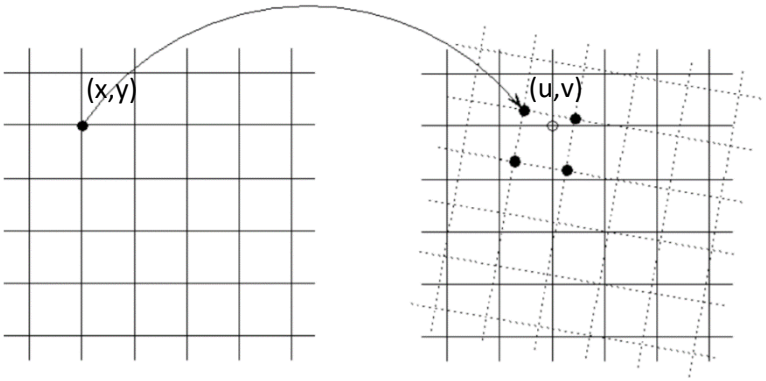
GEOMETRIC TRANSFORMATION

The operations which modify the spatial relationship between the pixels in an image are called Geometric transformation. These transformations mainly modify the position of pixels rather the intensities. Geometric transformations require 2 steps.

1. **Spatial Mapping**

   Spatial Mapping is transforming a point (x,y) in an image to the new coordinate system (u,v) by applying some functions. During this transformation, we may get points which are not integers giving a need for the 2nd step Interpolation

   $$u = f1(x, y)$$
   $$v = f2(x, y)$$



https://www.cis.rit.edu/class/simg782/lectures/lecture_02/lec782_05_02.pdf

2. **Interpolation**

   A process of estimating unknown values using the know values in called Interpolation.

Applications

Geometric transformation has a wide variety of application. Some of them are

1. Aligning images taken at different times is a structures way to make the things more organized.
2. They are used to correct images for lens distortion and correct effects of camera orientation.
3. They also play a key role in Image Morphing or creating special effects.
4. They are used in image pre-processing steps in applications such as document understanding, where the scanned image may be mis-aligned.

IMPLEMENTATION

GUI

The GUI is implemented using the Tkinter Package of Python.

Scaling

Scaling performs geometric transformation which can be used to shrink or zoom the size of an image. It changes the size of the image by changing the no. of pixels. Zooming is used to make the details of the image more clear and visible. Shrinking is mostly used to create thumbnail images.

To implement this, we take the input from the user, the factor which they want to scale the image in x and y directions along the interpolation technique they prefer to use. We have 4 different interpolations techniques namely Nearest Neighbor, Bilinear, Bicubic, Lanczos4.

Using the x and y factors provided by the user, a new image containing all 0s is created.

Width of the new image = int(original width * x-scale factor)

Height of the new image = int(original height * y-scale factor)

Now we traverse every cell in the new image and find the mapping in the original image.

Row mapping = int(current row / y-scale factor)

Column mapping = int(current column / x-scale factor)

This mapping is used to identify the pixel in the original image and with the help of interpolation technique, we find the value for the current cell.

Interpolation Techniques

Nearest Neighbor Interpolation

In this interpolation method, the position of a pixel in the resultant image (expanded or shrunk) is converted back into the original image. The intensity of this pixel is set to the intensity of the pixel nearest to it. This method is simple but discontinuous, thus has no regularity.

Bilinear Interpolation

Bilinear interpolation unlike nearest neighbor considers the closest 2x2 neighborhood pixel values surrounding the unknown pixel's location. First the linear interpolation is done along rows and then interpolation is done along the resultant column.

The linear interpolation in one direction is given by the expression:

$I = I1 + (I2 - I1) * (x - x1 / x2 - x1)$

Where x is a point in between two points x1 and x2 and I1 and I2 are their intensities respectively.

Bicubic Interpolation

Bicubic interpolation solves for the value at the unknown pixel's location by considering the 16-pixel values surrounding the interpolation region. First the horizontal cubic interpolation is done and then the vertical cubic interpolation. For the horizontal interpolation portion of this algorithm, a cubic must be defined for each row of the 4x4 pixel region.

$$V(x) = Ax^3 + Bx^2 + Cx + D$$

The values are inserted into the cubic and solved for each of the 4 pixels in the row. These 4 linear equations can be used to solve for the coefficients A, B, C and D.

After solving all the row interpolations, a vertical cubic interpolation is done through the row interpolation points. Using the same technique as with the row interpolations, the value of y is plugged into the cubic for each of the known row offsets and the coefficients are calculated. Thus, finding the intensity at the unknown pixel's location.

Lanczos 4 Interpolation

This method is based on the 4-lobed Lanczos window function as the interpolation function, hence termed as Lanczos order four interpolation. It is to be used as a low-pass filter to smoothly interpolate the value of a digital signal between its samples. It uses source image intensities at 64 pixels nearest to unknown pixel.

The interpolated at value x is obtained by the discrete convolution of those samples with the Lanczos kernel along each row and then along the resultant column. The Lanczoz kernel (L(x)) is a normalized sinc function multiplied by sinc window.

$$L(x) = \begin{cases} 1 & \text{if } x = 0, \\ \dfrac{a \sin(\pi x) \sin(\pi x/a)}{\pi^2 x^2} & \text{if } -a \leq x < a \text{ and } x \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Time Complexity Evaluation

Original size (hxw)	**New Size (hxw)**	**Interpolation**	**Time (Built-in) sec**	**Time (Our program) sec**
512x512	256x256	Nearest Neighbor	0.0	0.20
512x512	256x256	Bilinear	0.003	1.99
512x512	256x256	Cubic	0.003	4.08
512x512	256x256	Lanczos4	0.037	14.20
512x512	512x768	Nearest Neighbor	0.0	1.16
512x512	512x768	Bilinear	0.003	11.84
512x512	512x768	Cubic	0.003	22.25
512x512	512x768	Lanczos4	0.015	150.53
128x128	512x512	Nearest Neighbor	0.0	0.7966
128x128	512x512	Bilinear	0.0	7.2338

128x128	512x512	Cubic	0.0	33.3550
128x128	512x512	Lanczos4	0.0156	131.3345
128x128	1024x1024	Nearest Neighbor	0.0	3.2336
128x128	1024x1024	Bilinear	0.015	29.9864
128x128	1024x1024	Cubic	0.0	135.9492
128x128	1024x1024	Lanczos4	0.015	474.3521
128x128	1280x1280	Nearest Neighbor	0.003	4.2490
128x128	1280x1280	Bilinear	0.003	45.1465
128x128	1280x1280	Cubic	0.004	211.6101
128x128	1280x1280	Lanczos4	0.015	884.8461

Computational time in seconds using Intel® Core™ i5 CPU@2.50GHz and RAM of 8GB.

With the custom functions, Nearest Neighbor and Bilinear are faster compared to the Cubic and Lanczos4. When the size of the image increases, the time taken for Scaling increases. The built-in functions perform faster about 1400 times in case of Lanczos4 shrinking.

When we shrink or zoom the image by a very high factor, then the quality of the image deteriorates.

Image Quality Evaluation:

Interpolation Type	**Feeling**	**Edges**	**overall**
Nearest Neighbor	mosaic or jagged effect	jagged	worst
Bilinear	blur, not sharp	blur	poor
Bicubic	fuzzy, sharp	more clear edges	better
Lanczos 4	fuzzy, sharp (similar to bicubic)	ringing	better

Translation

Translation basically means that we are shifting the image by adding/subtracting the X and Y coordinates. There are two methods to perform this technique. One is mapping the position of each picture pixel in an input image into a new position in an output image. Another one is creating a new image as output and insert each image pixel from the original image to the new image.

$$x = x_0 + dx \qquad x_0 = x - dx$$
$$y = y_0 + dy \qquad y_0 = y - dy$$

This is the formula I used to perform the image translation. I extracted all the pixels I want and inserted them into a whole new image, which has the same size as the original image.

Result

Using X:100, Y: 50 for moving the original image to right-bottom and using X:-100, Y:-50 for moving the original image to top-left.

Time Complexity Evaluation

X, Y value	Time (Built-in) sec	Time (Our program)
X: 100, Y: 50	1.23	0.173
X: -100, Y: -50	1.20	0.177

Computational time in seconds using Intel® Core i7 2.5GHz and Memory 16GB

Rotation

Image rotation performs a geometric transform which maps the position (x, y) of the picture element in an input image into a position (x1, y1) in an output image by rotating it through a user-specified angle θ.

Method & implementation

Since rotating, the width and height of the image will change like Fig. 1. We need to recalculate the size of the new image, the formulas follow:

newWidth= width*cos(a)+height*sin(a);

newHeight=height*cos(a)+width*sin(a);

For each point, the geometry and formula to calculate the new coordinate is like Fig.3. However, we consider the image center as origin (x0, y0), and there are some offsets. We use two loops to get new coordinate which calculate by each point of the original image. When recalculate the new coordinate, we use bilinear interpolation to get the pixel for each coordinate. The formulas are:

Y= cos(a)*(y-(newWidth/2))-sin(a)*(x-newHeight/2)+width/2;

X= sin(a)*(y-(newWidth/2))+cos(a)*(x-newHeight/2)+height/2;

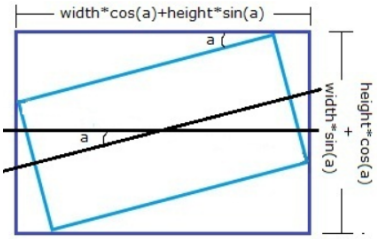


Fig.1 new image width & height

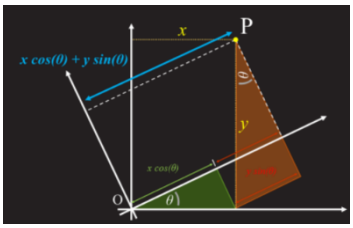


Fig. 2 rotation geometry

$$\begin{bmatrix} x^* \\ y^* \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Fig. 3 new coordinate

http://datagenetics.com/blog/august32013/index.html

Result &finding

Using angle=25 for clockwise and anti-clockwise.

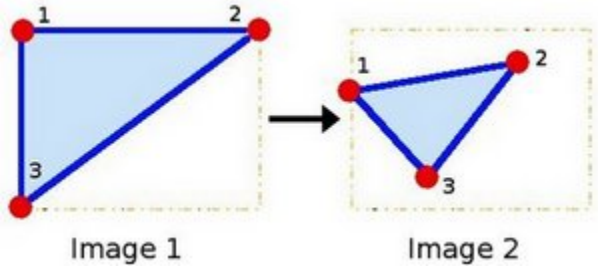
In this process, we found that we cannot just use cos=np.cos(angle)to calculate in OpenCV. The problem is that trigonometric function uses radians. So the correct statement is cos=np.cos(float (angle)*PI/180.). And the screen origin is the top leftmost, if we want to use the center of the image as origin, we must consider the offset for x-direction and y-direction.

Time Complexity Evaluation

Angle and orientation	Time (Built-in) sec	Time (Our program)
25, clockwise	1.241	9.231
25, anti-clockwise	1.239	9.428

Affine Transformation

Affine transformation uses transformation matrices to implement geometric transformations. Transformation matrices are 3x3 matrices that allow arbitrary linear transformations to be displayed in a consistent format, suitable for generic computation. In our implementation of affine, we take two different triangles on the image and create a transformation matrix that tries to fit one triangle into the other triangle.



https://docs.opencv.org/3.4/d4/d61/tutorial_warp_affine.html

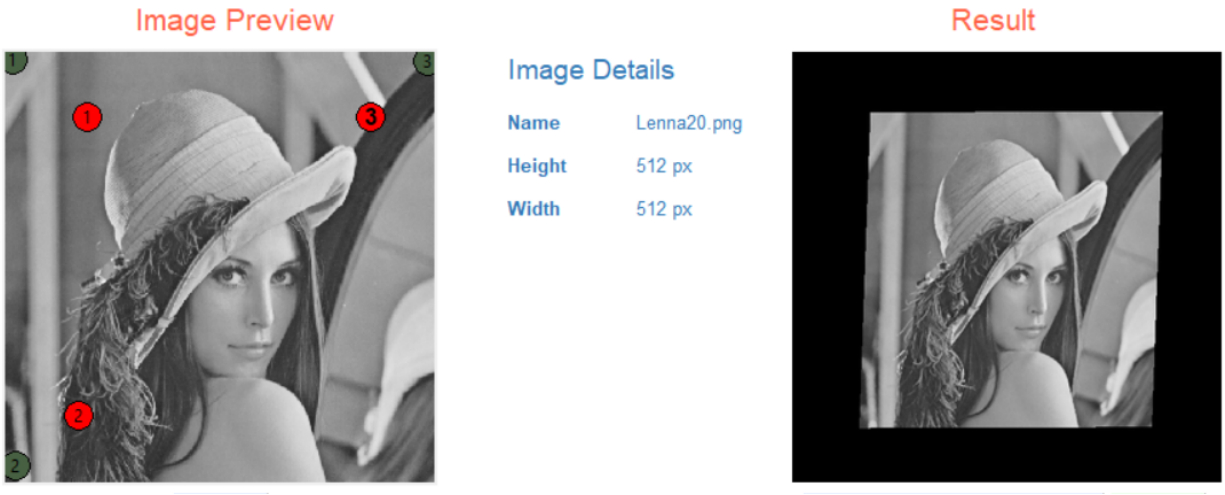
To do this, we apply linear algebra and solve for the unknowns. An example of one of the generated transformation matrices is as follows:

```
[[3.90130400e-01 1.01489801e-01 1.96137938e+02]
 [4.39478567e-02 4.79404104e-01 1.34332013e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

The affine transformation algorithm takes an image and the transformation matrix as an input and applies it to the following formula:

$$\left|\frac{x_2}{y_2}\right| = A \times \left|\frac{x_1}{y_1}\right| + B$$

Where x_1 and y_1 is the input image, x_2 and y_2 is the output image, and A and B is the transformation matrix. The following is an example of the result of affine transformations.



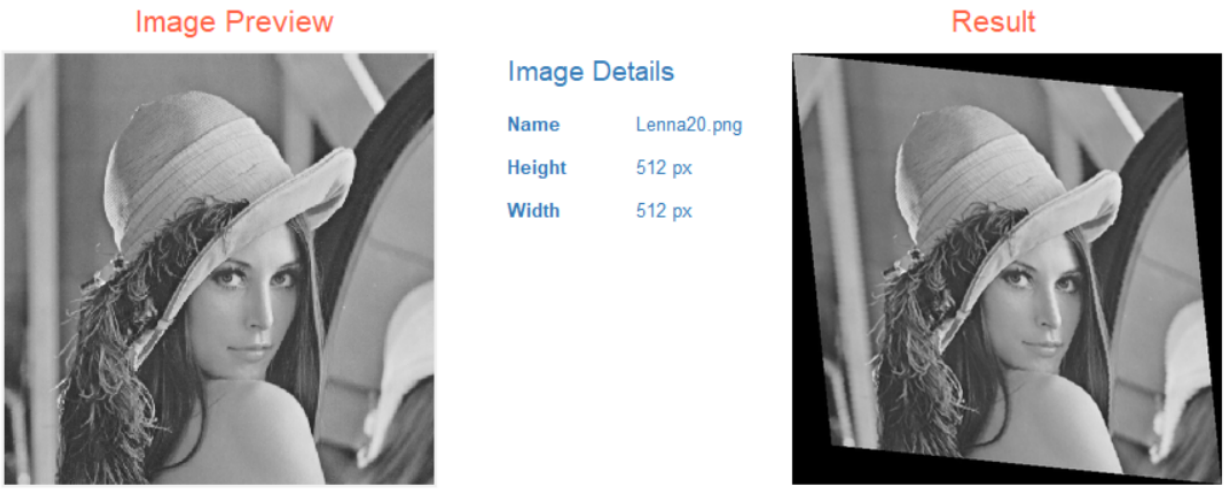
Shear

Affine transformation is directly used to shear an image. The following are transformation matrices that result in an image shear in the x direction and y direction respectively.

$$\begin{bmatrix} 1 & \tan\theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ \tan\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The following is an example of shearing in positive the x and y direction.

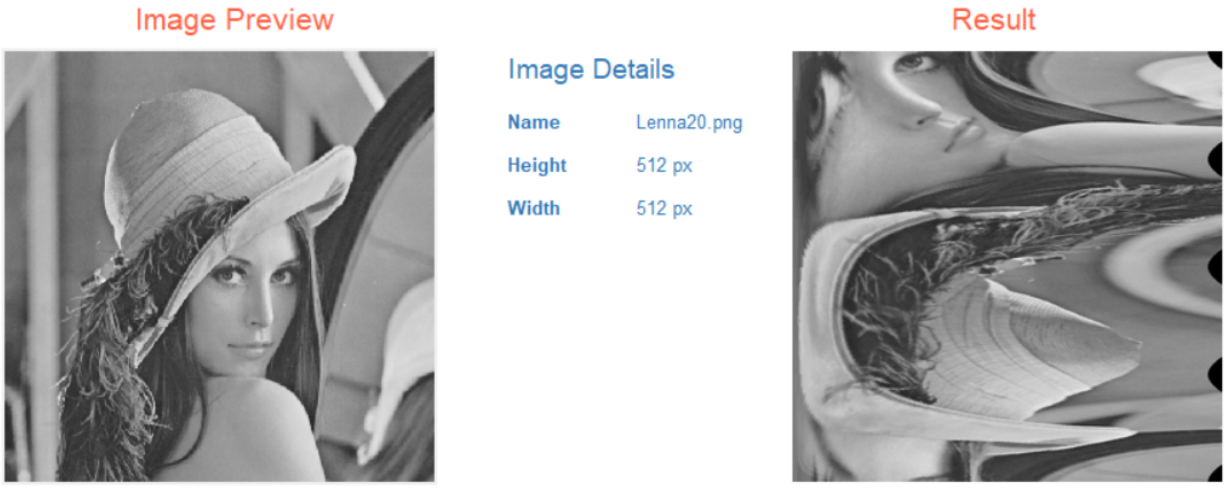


Polar Coordinates

The polar coordinate system is a two-dimensional coordinate system in which each point on an image (x, y) is determined by the distance from a reference point and an angle from a reference direction (r, φ). For our implementation we essentially remap an image to polar space where the x-axis is the radius, and the y-axis is the angle. Polar coordinates are usually used to describe domains in the plains with some rotational symmetry. The following is the formula we use to map Cartesian coordinates to polar:

$$\begin{matrix} r = \sqrt{x^2 + y^2} \\ \varphi = \text{atan}_2(y, x) \end{matrix} \leftrightarrow \begin{matrix} x = r\cos\varphi \\ y = r\sin\varphi \end{matrix}$$

The following is an example of polar coordinates where the reference point is the center:



Log-Polar Coordinates

Similar to polar coordinates, we remap an image to log-polar space. Instead of using raw distance from a reference point, we use the logarithm of the distance from the reference point. In areas like harmonic and complex analysis, log-polar coordinates are more useful than polar coordinates. The following is the formula we use to map Cartesian coordinates to polar:

$$\begin{matrix} \rho = \log\sqrt{x^2 + y^2} \\ \theta = \arctan(y/x) \ \ if \ x > 0 \end{matrix} \leftrightarrow \begin{matrix} x = e^{\rho}\cos\theta \\ y = e^{\rho}\sin\theta \end{matrix}$$

The following is an example of log-polar coordinates where the reference point is the center: