COMS 3251 Spring 2021: Lab 2

YOUR NAME(s): Vicente Farias

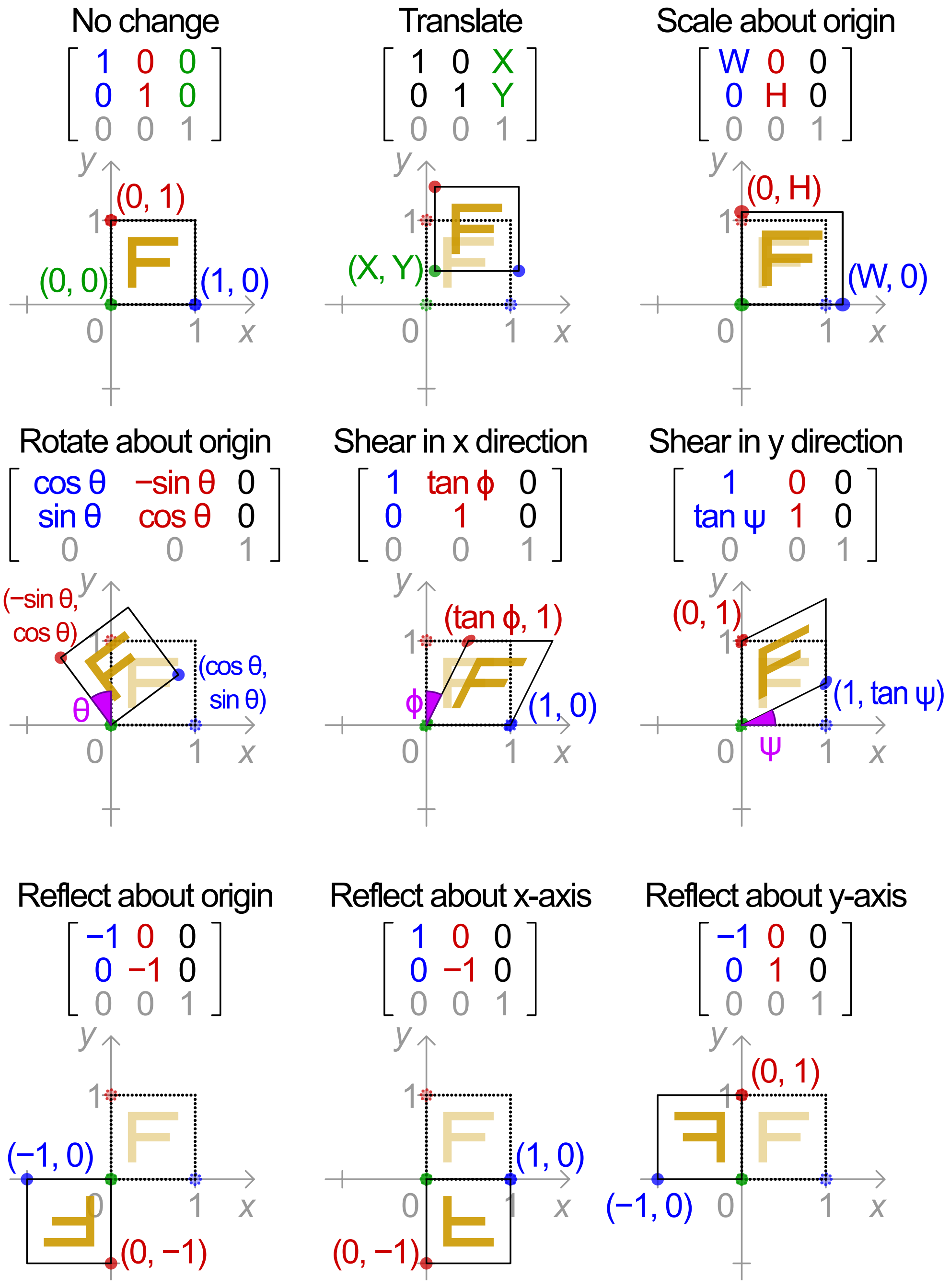
YOUR UNI(s): vf2272

Affine Transformations

We have seen that linear functions can be represented as matrix-vector multiplications. Certain linear functions also have a geometric interpretation; we have seen examples of reflections and rotations in $\mathbb{R}^2$. On the other hand, one kind of transformation that is not linear is **translation**, or the addition of a constant vector.

We cannot represent vector addition by an equivalent matrix-vector multiplication, at least in the usual sense. What we need to do is extend our vectors using **homogeneous coordinates**. We take a 2-vector $[x, y]$ and recast it as $[x, y, 1]$ (simply add a 1 in a third coordinate). We will see that we can now perform **affine transformations**, which will include linear transformations and translations, as matrix-vector multiplication.

Examples



Some examples of affine transformations in $\mathbb{R}^2$ are shown in the graphic above, taken from Wikipedia's page on transformation matrices. Notice that many of them look similar to the linear transformations that we've seen; the original 2D matrices appear in the upper left 2x2 submatrix, while the last row and last column mostly look like $[0, 0, 1]$. You should convince yourself that multiplying a vector in homogeneous coordinates (1 in the third slot) by an affine transformation matrix leaves you with another vector in homogeneous coordinates.

Take translation as an example. Starting with the vector $[\mathbf{u}, \mathbf{v}, 1]$, multiplication with the translation matrix gives us

$$\begin{bmatrix} 1 & 0 & \mathbf{x} \\ 0 & 1 & \mathbf{y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{u} + \mathbf{x} \\ \mathbf{v} + \mathbf{y} \\ 1 \end{bmatrix}.$$

This achieves the task of translation, or vector addition for the two coordinates that we care about! All thanks to the presence of the 1 in that third slot.

Note that the rotation and shears shown above measure the angles with respect to the y axis, in contrast to what we saw in lecture. To keep things simple, you should implement the matrices shown here in the first exercise below. Since OpenCV has its y axis pointing downward, the angle direction reversal is negated, and we can still associate counterclockwise rotations with positive angles and clockwise rotations with negative angles (in short, just follow what's shown here and everything will work out!).

Composition

As with regular linear transformations in $\mathbb{R}^2$, composition of affine transformations can be achieved by multiplication of the aforementioned matrices. Note that the composition is also guaranteed to be an affine transformation. The multiplication of two matrices with $[0, 0, 1]$ in the last row produces a matrix that also has $[0, 0, 1]$ in the last row.

As a reminder, matrix multiplication is not commutative, so ordering matters. If we want to apply the affine transformations A, B, and C in that order, the composite transformation is given by $D = CBA$. The matrix-vector product $D\mathbf{x}$ would then be the result of the composite transformation applied to $\mathbf{x}$.

OpenCV

You'll be using some basic functionalities of the OpenCV package in this lab. It contains a number of standard algorithms for computer vision and machine learning (although you won't need to worry about them at this point).

The convention for representing images with OpenCV has the origin at the top left corner of the image. The x axis increases moving rightward, and the y axis increases moving downward.

alt text

Exercises

For the rest of the lab session, work on the following exercises with your lab partner. We recommend that you work on the written question together. For the programming problems, one way to proceed would be to discuss a solution strategy together and then split up the implementations between the two of you.

In [ ]:

```
### THIS NEEDS TO BE RUN FIRST ###
import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
```

PART 1 (15 points)

Your first task is to implement the following functions to return the specified affine transformation matrices as NumPy arrays. Each function takes in inputs specifying the amount or angle of the transformation. Remember that many common mathematical functions can be called using NumPy.

In [ ]:

```
### PART 1 ###
# Complete the functions below

def get_translate(x, y):
  """
  Inputs:
  x, x-axis translation in pixels
  y, y-axis translation in pixels

  Output: A 3x3 numpy array
  """
  ### REPLACE THE RETURN VALUE ###

  transform = np.array([[1, 0, x],
                        [0, 1, y],
                        [0, 0, 1]])

  return transform


def get_scale(w, h):
  """
  Inputs:
  w, x-axis scale (1 = 100%)
  h, y-axis scale (1 = 100%)

  Output: A 3x3 numpy array
  """
  ### REPLACE THE RETURN VALUE ###

  transform = np.array([[w, 0, 0],
                        [0, h, 0],
                        [0, 0, 1]])

  return transform


def get_rotate(theta):
  """
  Input: theta, rotation in radians
  Output: A 3x3 numpy array
  """
  ### REPLACE THE RETURN VALUE ###

  transform = np.array([[np.cos(theta), np.sin(theta), 0],
                        [-1 * np.sin(theta), np.cos(theta), 0],
                        [0, 0, 1]])

  return transform


def get_xshear(phi):
  """
  Inputs: phi, x-axis shear angle in radians
  Output: A 3x3 numpy array
  """
  ### REPLACE THE RETURN VALUE ###

  transform = np.array([[1, np.tan(phi), 0],
                        [0, 1, 0],
                        [0, 0, 1]])

  return transform


def get_yshear(psi):
  """
  Inputs: psi, y-axis shear angle in radians
  Output: A 3x3 numpy array
  """
  ### REPLACE THE RETURN VALUE ###

  transform = np.array([[1, 0, 0],
                        [np.tan(psi), 1, 0],
                        [0, 0, 1]])

  return transform
```

PART 2 (15 points)

As we noted, composition of affine transformations is not commutative in general. You will investigate the ordering of two particular transformations specified by your TAs. First complete the two functions below (they may not use all input arguments, depending on the transformations specified). They should return the composite affine transformation matrices, each computed in a different order.

Then input the parameters (also specified by your TAs) in the first line of the following code cell, run it, and examine the results. The red text image near the origin is the original. The blue and green images are the results of the two composite transformations.

In [134…

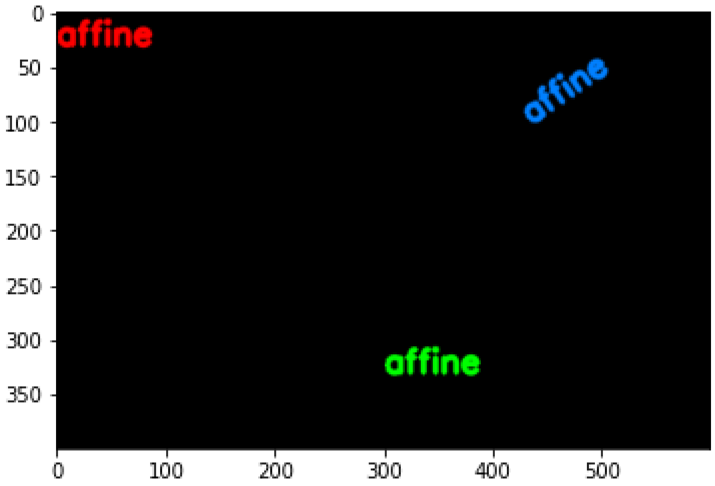
```
### PART 2 ###
# Complete the functions below

def funcA(x, y, alpha, beta):
  """
  Inputs:
  x, y: Amounts in pixels
  alpha, beta: Angles in radians

  Output: A 3x3 numpy array
  """
  ### REPLACE THE RETURN VALUE ###

  rotation = get_rotate(-1 * beta)
  translation = get_translate(x, y)

  transform = np.matmul(translation, rotation)

  return transform


def funcB(x, y, alpha, beta):
  """
  Inputs:
  x, y: Amounts in pixels
  alpha, beta: Angles in radians

  Output: A 3x3 numpy array
  """
  ### REPLACE THE RETURN VALUE ###

  translation = get_translate(x, y)
  rotation = get_rotate(alpha)

  transform = np.matmul(rotation, translation)

  return transform
```

In [137…

```
# REPLACE THE PARAMETERS SPECIFIED BY YOUR TAS HERE
x, y, alpha, beta = (300, 300, 0.6, 0)

font = cv2.FONT_HERSHEY_SIMPLEX
text0 = cv2.putText(np.zeros((400,600,3), np.uint8), 'affine', (0,30), font, 1, (255,
text1 = cv2.putText(np.zeros((400,600,3), np.uint8), 'affine', (0,30), font, 1, (0,255
text2 = cv2.putText(np.zeros((400,600,3), np.uint8), 'affine', (0,30), font, 1, (0,128

A = funcA(x, y, alpha, beta).astype(np.float32)
B = funcB(x, y, alpha, beta).astype(np.float32)
text1_t = cv2.warpAffine(text1, A[:2,:], (600, 400))
text2_t = cv2.warpAffine(text2, B[:2,:], (600, 400))
plt.imshow(text0 + text1_t + text2_t)
```

Out[137]:

```
<matplotlib.image.AxesImage at 0x7ffb08157c50>
```



Explain how the two results are obtained and why they are different from the perspective of the geometric transformation. Do not simply reference the non-commutativity of matrix multiplication. Instead, try to include some intuition about what the individual transformations do.

Though both operations involve the same transformation, the second operation includes a rotation by 0.6 radians anticlockwise about the origin. Therefore, the resulting blue text is not in the same location as the green text.

Even if the operations rotated by the same angle in the same direction and translated by the same amount, the resulting text would not be in the same place because of the ordering of the transformations. The rotation rotates the text on a circle whose radius starts at the origin and ends at the text. By translating first, the text is moved to a larger circle, and the rotation moves it further. By rotating first, the text moves a smaller amount, and is then translated to a different location than the other transformation.

PART 3 (20 points)

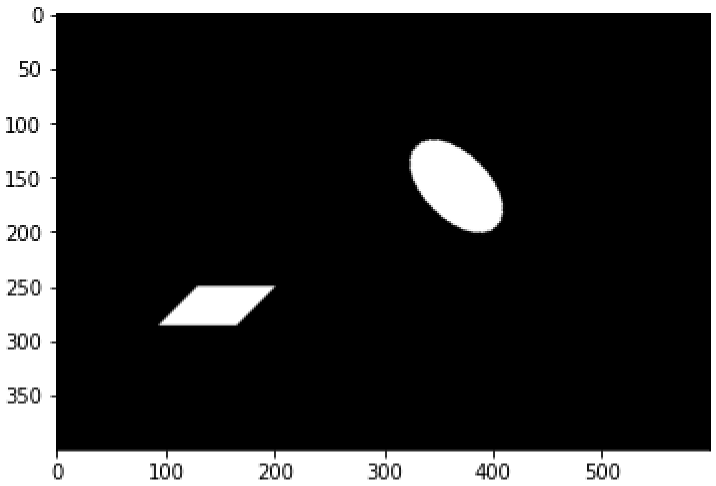
In this last part, you will try to reconstruct a given image using some combination of affine transformations. You will first be provided with a "puzzle.png" image file. Upload this file to your Colab workspace by clicking on folder icon on the left sidebar and clicking "Upload". Then run the code cell below.

In [ ]:

```python
# If your image was uploaded successfully, the template will be shown below
template = cv2.imread('puzzle.png', cv2.IMREAD_UNCHANGED)
if template is None:
        print("ERROR: upload the image!")
plt.imshow(template)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7ffb08425b00>
```



If the file was imported successfully, you should see the image that you will try to reconstruct. There are two separate transformations, one starting with a circle and the other starting with a square, both near the origin. Your task is to complete the functions below to successfully transform the original circle and square into their resultant shapes in the image file. As in Part 2, they will consist of some composition of the basic transformations you implemented in Part 1.

Hints

To prevent this task from becoming manual unconstrained optimization in continuous space, we have discretized this problem as follows:

- All **translation** was done in steps of 50 pixels.
- All **rotations** were done in steps of $\pi/4$.
- All **shears** were done in steps of $\pi/4$.
- All **scalings** were done in steps of 0.25.
- Only **horizontal** shearing and stretching was used.
- The circle wasn't sheared.
- You should probably apply the translation matrix **last**.
- It's best to just start trying out some transformations to see what they look like before trying to solve the puzzle! Just running everything will give you the shapes without any transformations.

In [ ]:

```
### PART 3 ###
# Complete the functions below

def get_circle_transform():
  """
  Generate the final affine transformation matrix for the circle.
  Returns a 3x3 numpy array
  """

  ### REPLACE THE RETURN VALUE ###

  translate = get_translate(307.5, 142.5)
  scale = get_scale(1, 1.75)
  rotate = get_rotate(math.pi/4)

  transform = np.matmul(rotate, scale)
  transform = np.matmul(translate, transform)

  return transform

def get_square_transform():
  """
  Generate the final affine transformation matrix for the square.
  Returns a 3x3 numpy array
  """

  ### REPLACE THE RETURN VALUE ###

  scale = get_scale(1.5, 0.75)
  shear = get_xshear(-1 * math.pi / 4)
  translate = get_translate(125, 250)

  # transform = scale

  transform = np.matmul(shear, scale)
  transform = np.matmul(translate, transform)

  return transform
```

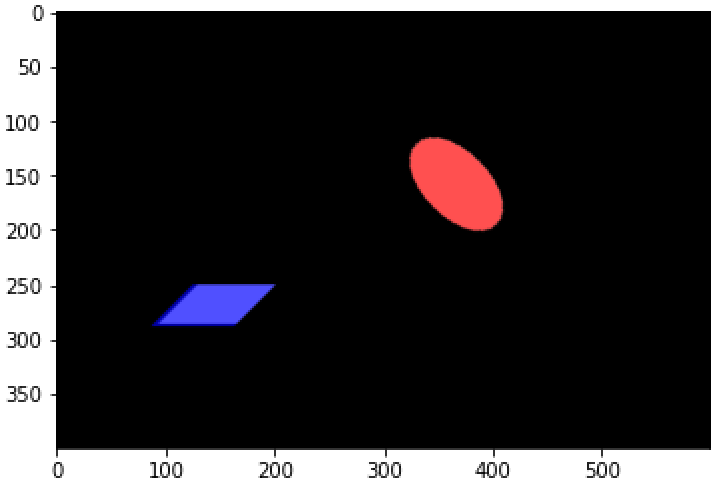
Once you are ready to test your transformations (or if you just want to see the default behavior), run the code cell below. Your transformed shapes will be drawn in blue and red, with some transparency. If the results match exactly, there will be no visible white borders inside or outside the shape.

In [ ]:

```
def combine_layers(layers):
        layer0 = layers[0]
        for l in layers[1:]:
                layer0[:,:,0] = (l[:,:,3] / 255) * l[:,:,0] + ((255 - l[:,:,3]) / 255)
                layer0[:,:,1] = (l[:,:,3] / 255) * l[:,:,1] + ((255 - l[:,:,3]) / 255)
                layer0[:,:,2] = (l[:,:,3] / 255) * l[:,:,2] + ((255 - l[:,:,3]) / 255)
        return layer0

puzzle = template.copy()
circ_M = get_circle_transform().astype(np.float32)
sq_M = get_square_transform().astype(np.float32)

bg1 = np.zeros((400, 600, 4))
sq = cv2.rectangle(bg1, (0,0), (50,50), (0,0,255,175), -1)
sq_t = cv2.warpAffine(sq, sq_M[:2,:], (600, 400))

bg2 = np.zeros((400, 600, 4))
circ = cv2.circle(bg2, (30,30), 30, (255,0,0,175), -1)
circ_t = cv2.warpAffine(circ, circ_M[:2,:], (600, 400))

output = combine_layers([puzzle, sq_t, circ_t])
plt.imshow(output)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7f85226d1748>
```


0
50
100
150
200
250
300
350
0
100
200
300
400
500