**CS473 Assignment #2**

**Due 2/17/23**

**Basic Image formation and transformation in computer vision.**

**By Alif Jakir & Evan Couchman**

**Part 1**

1. *In class, we saw how to compute the inverse of the scaling matrix, not by a linear algebra approach but by intuiting what it means to scale downwards (shrinking is the multiplicative inverse of enlarging). Similarly, compute the inverse of a translation matrix. Hint: think about what it means to invert translation. We saw that inverting scaling means performing the opposite of the particular scaling transform: if you were enlarging the image to get the transformed image, then the image itself is a shrunk version of the transformed image, and vice versa. Apply a similar intuition for translation. (5 points)*

Inverse of a translation matrix:

A translation matrix is of the form:

|1 0 t\_x|

|0 1 t\_y|

|0 0 1 |

Thus (t\_x, t\_y) represents the translation vector. The inverse of this matrix is obtained by negating the translation vector, which means:

|1 0 -t\_x|

|0 1 -t\_y|

|0 0 1 |

In OpenCV and Python, we can implement this as follows:

```

import numpy as np

# translation vector

tx, ty = 10, 20

# translation matrix

T = np.array([[1, 0, tx], [0, 1, ty], [0, 0, 1]])

# inverse of the translation matrix

T\_inv = np.array([[1, 0, -tx], [0, 1, -ty], [0, 0, 1]])

```

1. *Using the same intuitions as what you used above, invert a rotation matrix. There is a fundamental relationship between a rotation matrix and its inverse. Can you identify this relationship? (5 points)*

Inverse of a rotation matrix:

A 2D rotation matrix is of the form:

|R(theta) 0|

|0 1|

where R(theta) is the 2x2 rotation matrix for an angle theta. The inverse of this matrix is simply the transpose of the matrix,

|R(theta)' 0|

|0 1|

where R(theta)' is the transpose of the 2x2 rotation matrix. In other words, the inverse of a rotation matrix is equal to its transpose. This is because rotation matrices are orthogonal matrices and their inverses are equal to their transposes.

Some code for this in Python:

```

import numpy as np

import math

# rotation angle (in degrees)

theta = 45

# convert degrees to radians

theta\_rad = math.radians(theta)

# rotation matrix

R = np.array([[math.cos(theta\_rad), -math.sin(theta\_rad), 0],

[math.sin(theta\_rad), math.cos(theta\_rad), 0],

[0, 0, 1]])

# inverse of the rotation matrix

R\_inv = R.T

```

1. *What should the inverse of a reflection matrix be, and why? (5 points)*

Inverse of a reflection matrix:

A reflection matrix is of the form:

| -1 0 |

| 0 1 |

The inverse of this matrix is simply the matrix itself, i.e.,

| -1 0 |

| 0 1 |

This is because the reflection matrix is its own inverse.

Code:

```

import numpy as np

# reflection matrix

R = np.array([[-1, 0], [0, 1]])

# inverse of the reflection matrix

R\_inv = R

```

1. *The inverse for the shear matrix is slightly trickier to intuit, so we will first calculate its inverse using a standard matrix-inversion formula. You will only need the inversion formula for a 2 ×2 matrix:*

[a b

c d]−1 = 1/(ad −bc) \*[ d −b −c a]

Use the above formula to calculate the inverse of the matrix for the shear transform

along the x-direction. Intuitively, what does this inverse mean? Can you obtain a simi-

lar formula for the inverse of the shear transform along the y-direction? (5 points).

Inverse of a shear matrix:

A 2D shear matrix along the x-direction is of the form:

|1 s\_x 0|

|0 1 0|

|0 0 1|

where s\_x is the shear factor along the x-axis. The inverse of this matrix can be obtained using the formula given in the question:

|1 -s\_x 0|

|0 1 0|

|0 0 1| / (1 - s\_x \* 0)

= |1 -s\_x 0|

|0 1 0|

|0 0 1|

which is the same as the original matrix, i.e., the inverse of the shear matrix along the x-axis is equal to the matrix itself.

A 2D shear matrix along the y-direction is of the form:

|1 0 0|

|s\_y 1 0|

|0 0 1|

where s\_y is the shear factor along the y-axis. To obtain the inverse of this matrix, we can again use the formula given in the question:

|1 0 0|

|-s\_y 1 0|

|0 0 1| / (1 - 0 \* s\_y)

= |1 0 0|

|-s\_y 1 0|

|0 0 1|

which is the inverse of the shear matrix along the y-axis.

Code:

```

import numpy as np

# shear factor along the x-axis

s\_x = 2

# shear matrix along the x-axis

Sx = np.array([[1, s\_x, 0], [0, 1, 0], [0, 0, 1]])

# inverse of the shear matrix along the x-axis

Sx\_inv = np.array([[1, -s\_x, 0], [0, 1, 0], [0, 0, 1]])

# shear factor along the y-axis

s\_y = 3

# shear matrix along the y-axis

Sy = np.array([[1, 0, 0], [s\_y, 1, 0], [0, 0, 1]])

# inverse of the shear matrix along the y-axis

Sy\_inv = np.array([[1, 0, 0], [-s\_y, 1, 0], [0, 0, 1]])

```

**Part 2**

Here are the original pictures:

****

** (not to scale)**

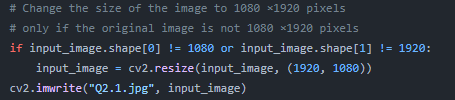
We turn all the images into grayscale images for easier manipulation.

We reformatted and utilized the function *transformImage()* into python using OpenCV and Numpy. We also created two other functions for ease of use when making a transformation function: *homography\_matrix()* & *transform\_matrix().*  *homography\_matrix()* allows for the user to fully input whatever numerical values they would like for a matrix [ABC] [DEF][GHI], while *transform\_matrix()* allows for easy curation of an affine transform matrix depending on the certain affine transforms the user would like to make.

The Python function *transformImage()* takes an input image (*I*), a transformation matrix (*A*), a transformation type (*transform\_type*), and an output image name (*output\_image\_name*) as arguments. The function first checks if the transform\_type is valid (either 'scaling', 'translation', 'rotation', 'reflection', 'homography', or 'affine') and raises a ValueError if it is not. Next, the function extracts the dimensions of the input image and defines the four corners of the input image as c1, c2, c3, and c4. The corners are then transformed using the homography matrix A into correspondences cp1, cp2, cp3, and cp4. If the transformation type is 'homography', the function also checks if the matrix is a homography matrix and, if it is, extracts the x and y coordinates of the transformed corners. The minimum and maximum x and y values of the transformed corners are then found, and a grid of x and y coordinates is created in the output image. The homography matrix is inverted to map points from the output image to the input image, and the homogenized output image coordinates are then mapped to input image coordinates using the inverted homography matrix. The x and y coordinates are then reshaped into a matrix and the input image is interpolated at the mapped coordinates to create the output image. If the transformation type is 'reflection', the output image is cropped to fit within the original image size. Finally, the function saves the input and output images to disk with the specified names.

*1. Change the size of the image to 1080 ×1920 (make sure the image you start with is not*

*already 1080 ×1920).*

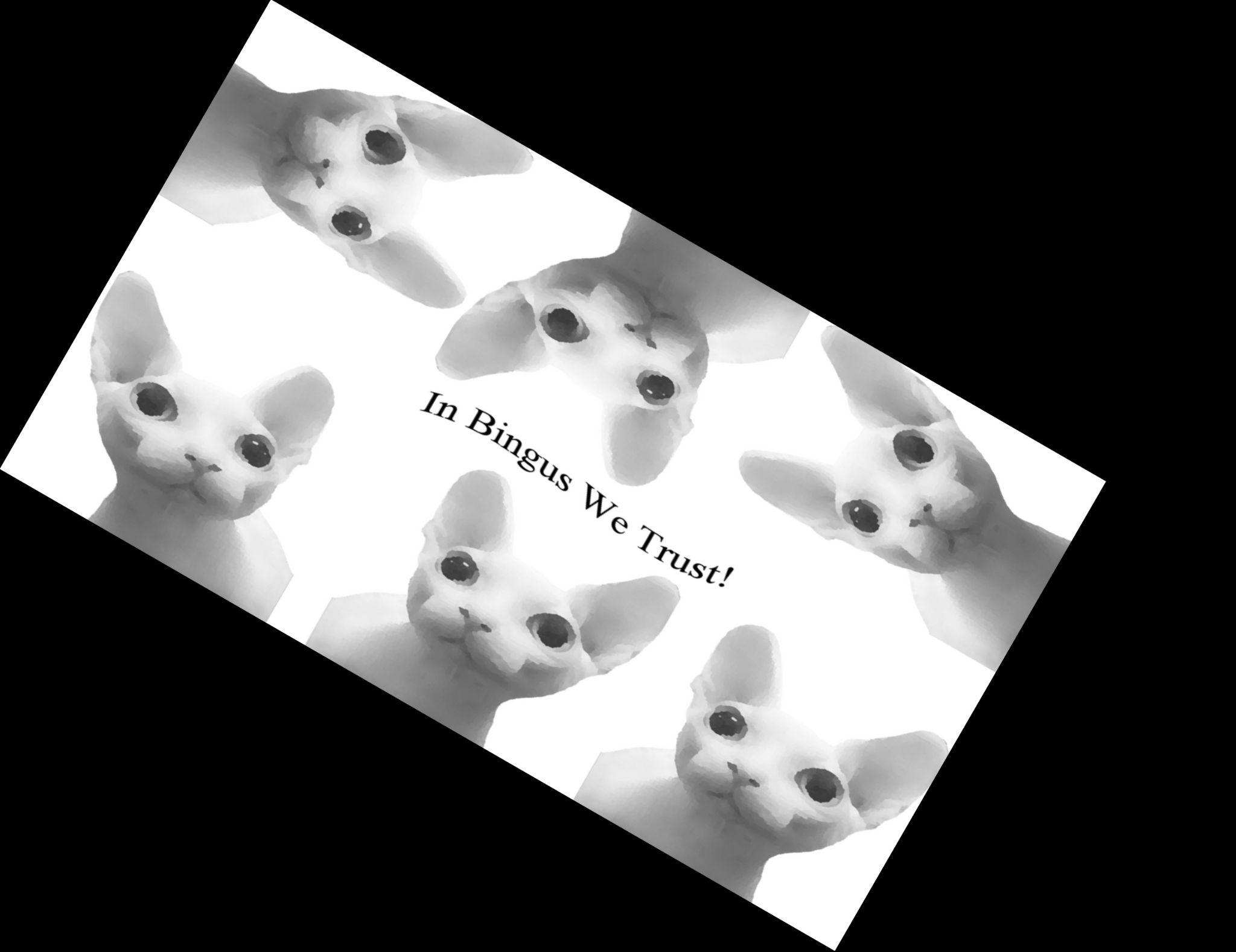
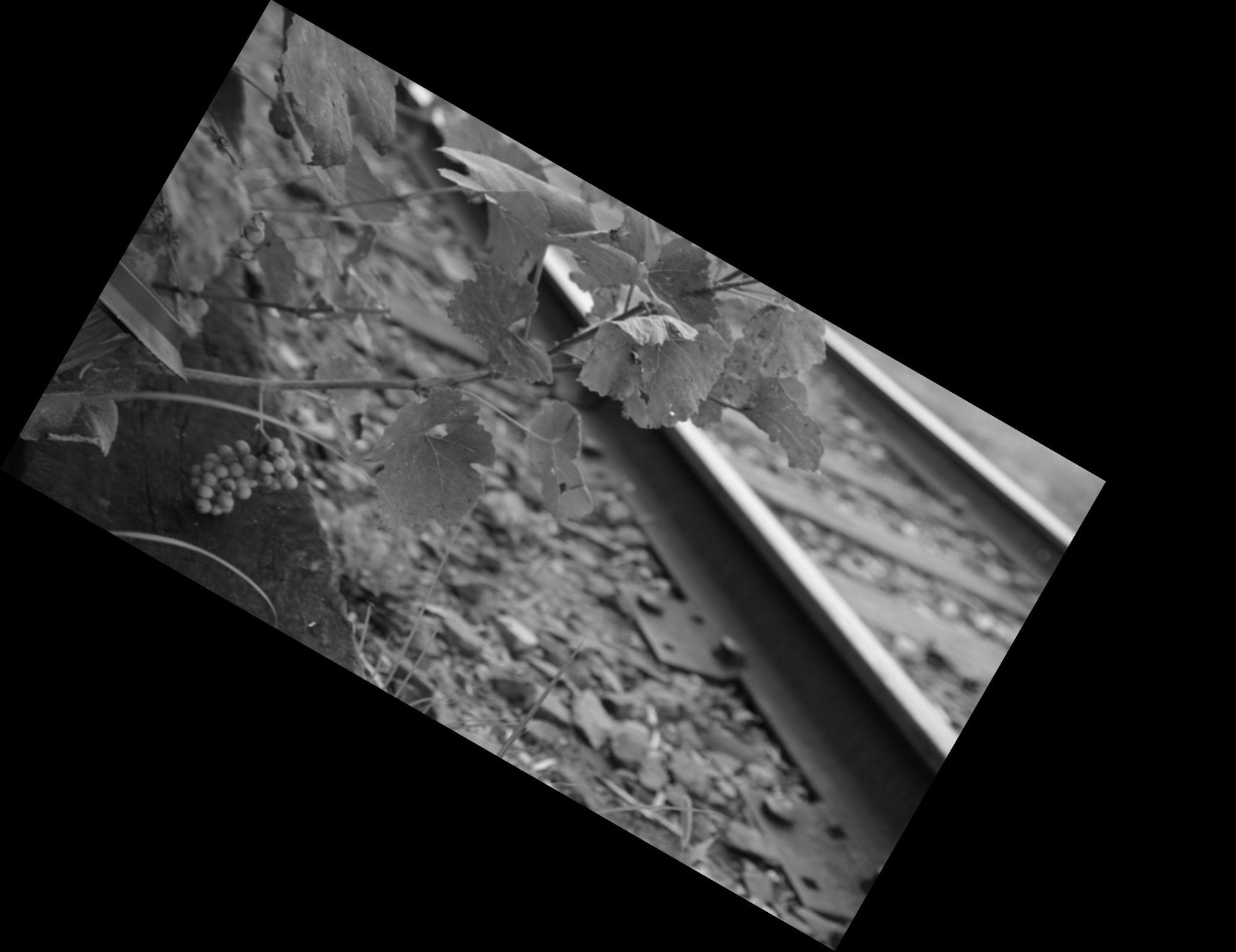
Here is the code utilized to do this task:******

*2. We reflect the images in the y direction*.

Here is the function calls utilized to do this task:****

*3. Rotate the image clockwise by 30 degrees*.

Here is the function calls utilized to do this task:

****

*4. Shear the image in the x-direction so that the additional amount added to each x value*

*is 0.5 times each y value.*

Here is the function calls utilized to do this task:

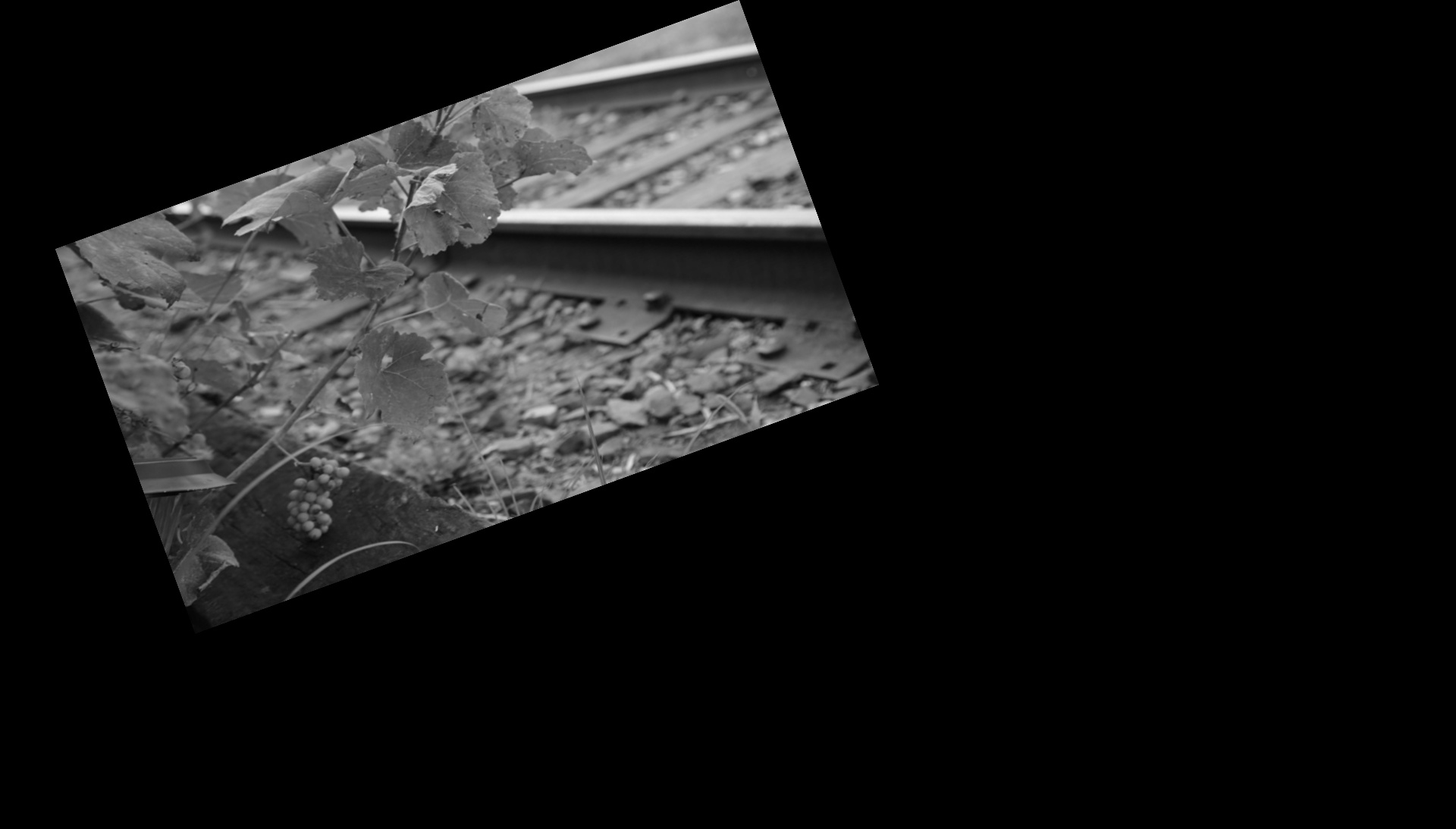
*5. Translate the image by 300 in the x-direction and 500 in the y-direction, then rotate the*

*resulting image counterclockwise by 20 degrees, then scale the resulting image down to*

*one-half its size. You should apply the transformImage function only once to do this.*

Here is the function calls utilized to do this task:





****

*6. The following two affine transforms:*

The first 3 images are of the transform matrix:

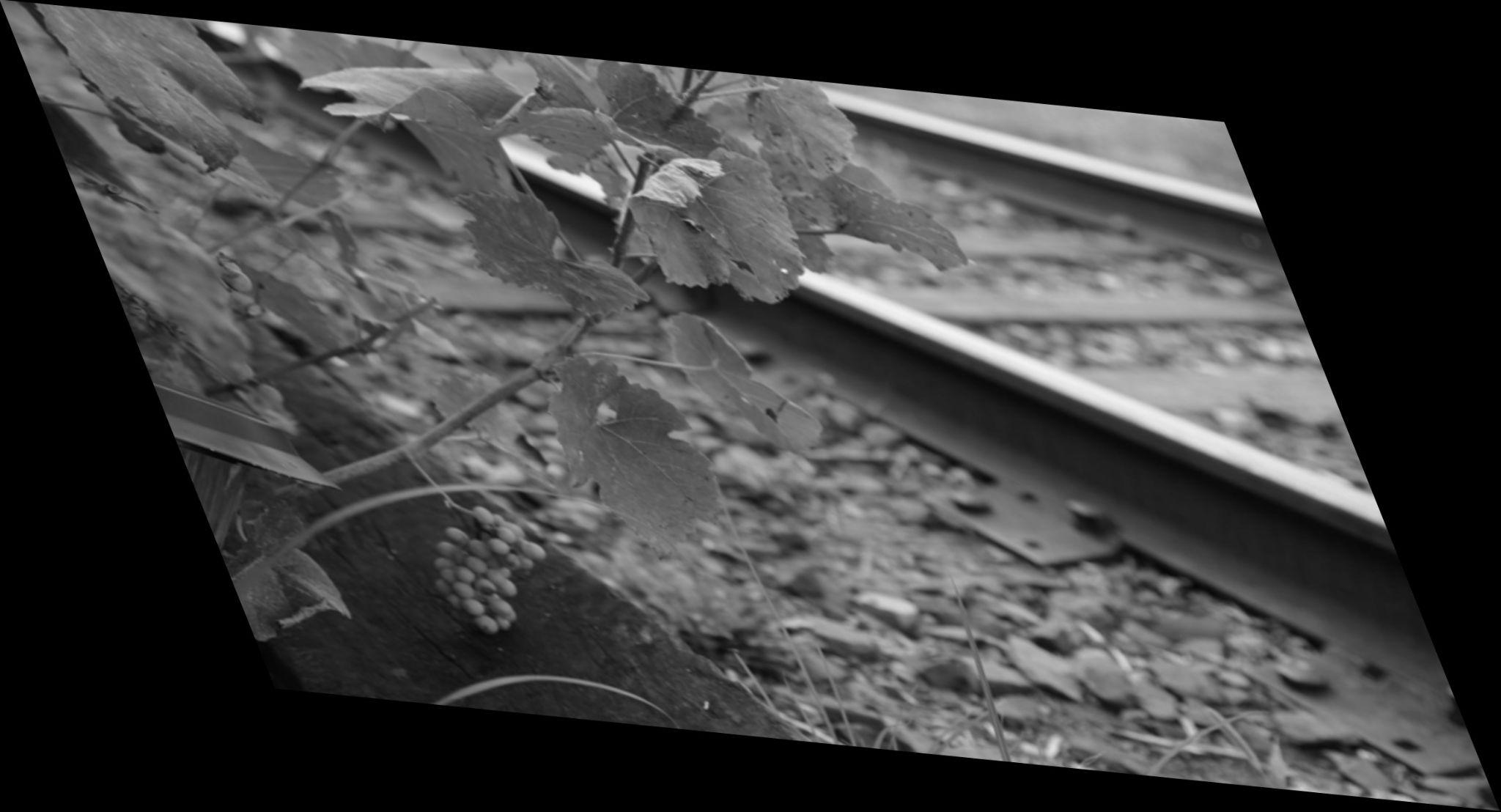
[1 0.4 0.4]

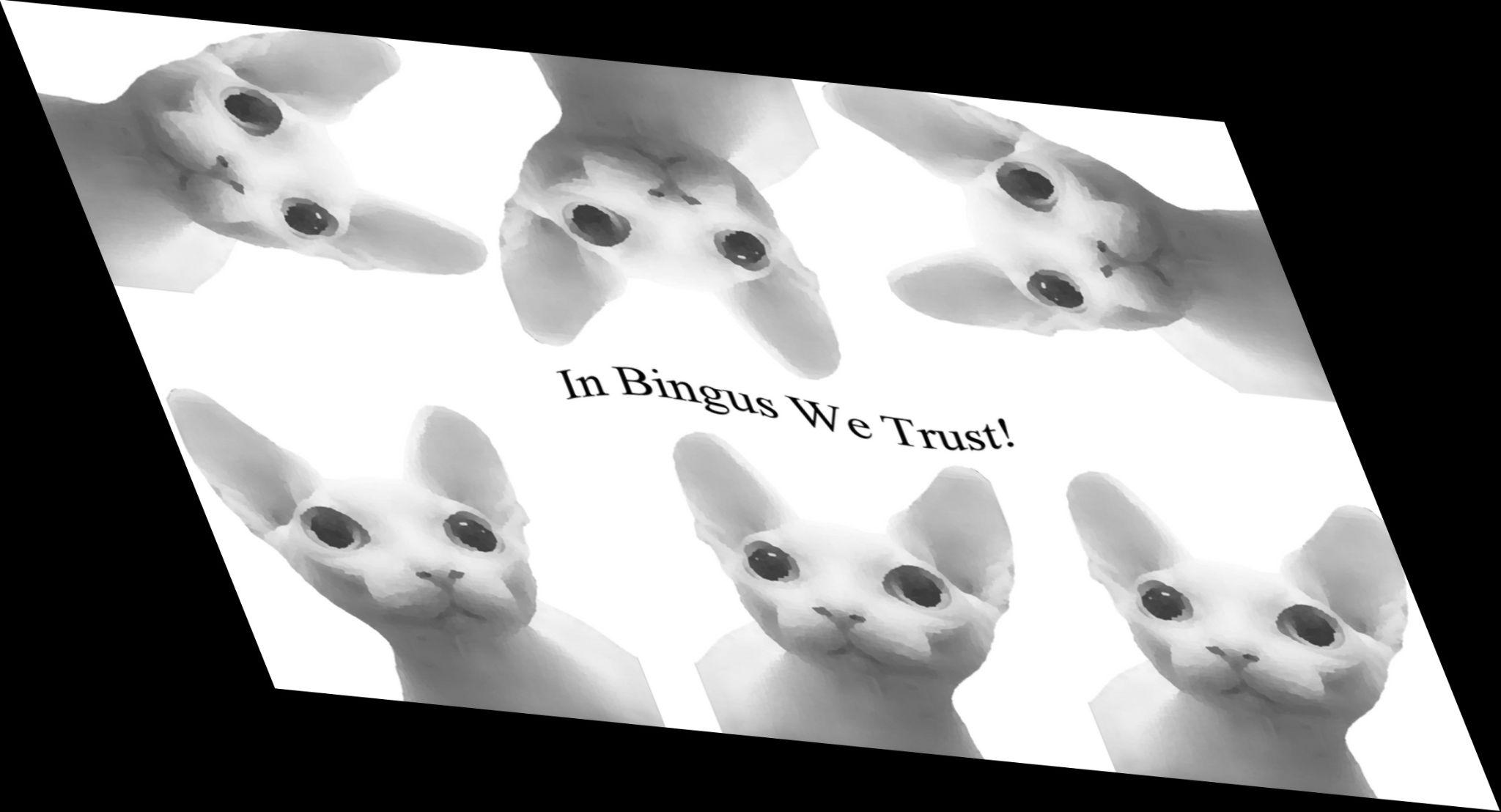
[0.1 1 0.3]

[0 0 1].

Here is the function calls utilized to do this task:







This next three images below are of the transform matrix:

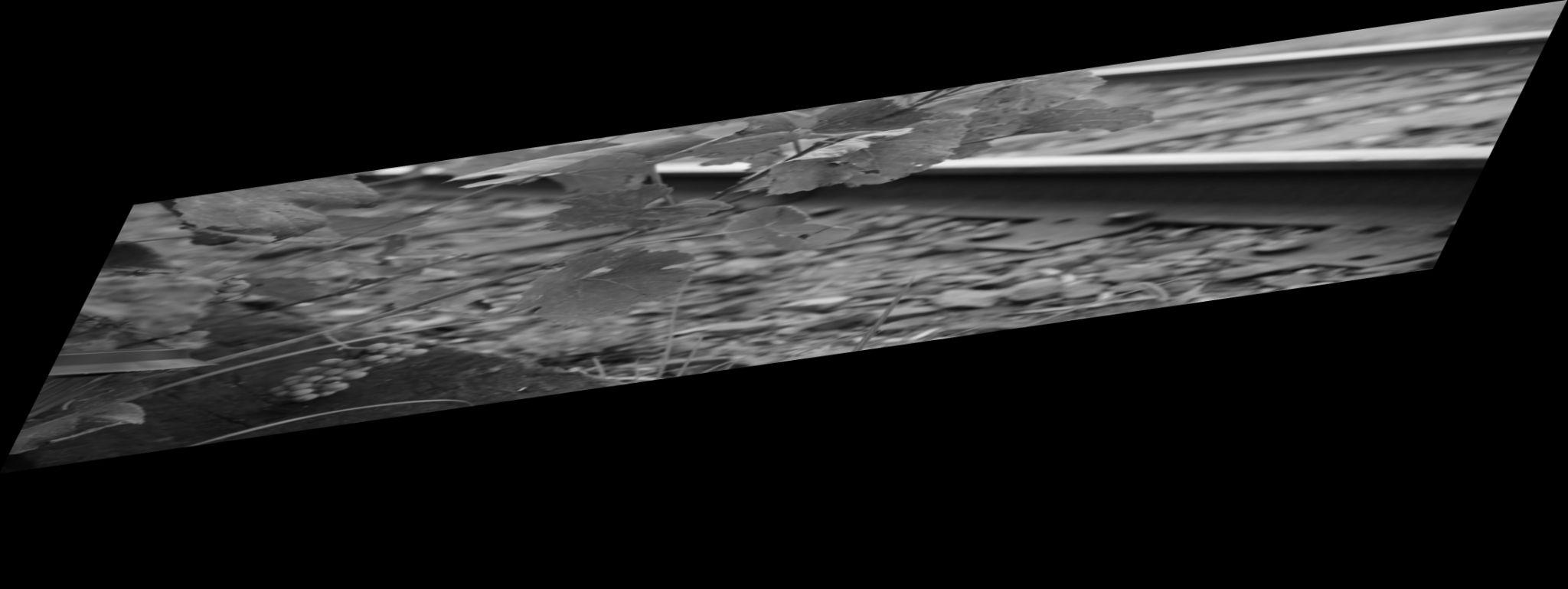
[ 2.1 -.35 −.1]

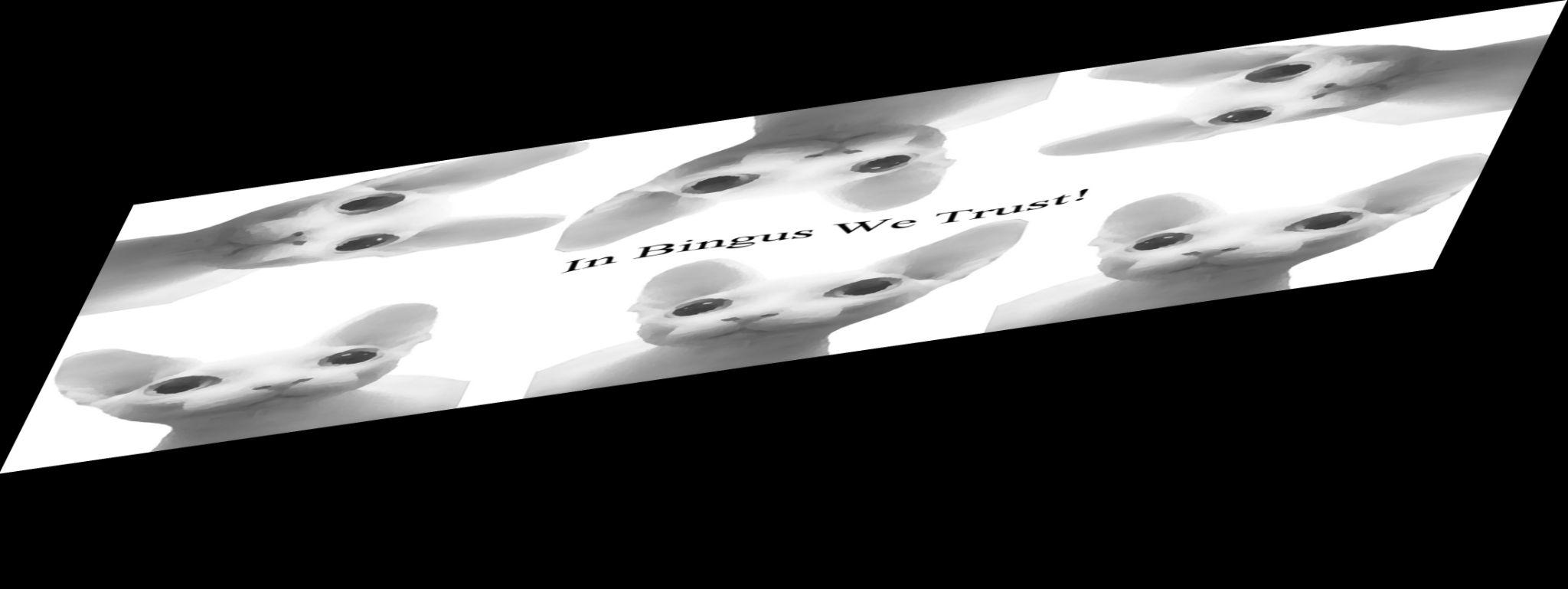
[−.3 .7 .3]

[0 0 1]

Here is the function calls utilized to do this task:



****

****

*7. The following two homography transforms:*

The first 3 images are of the transform matrices :

[.8 .2 .3]

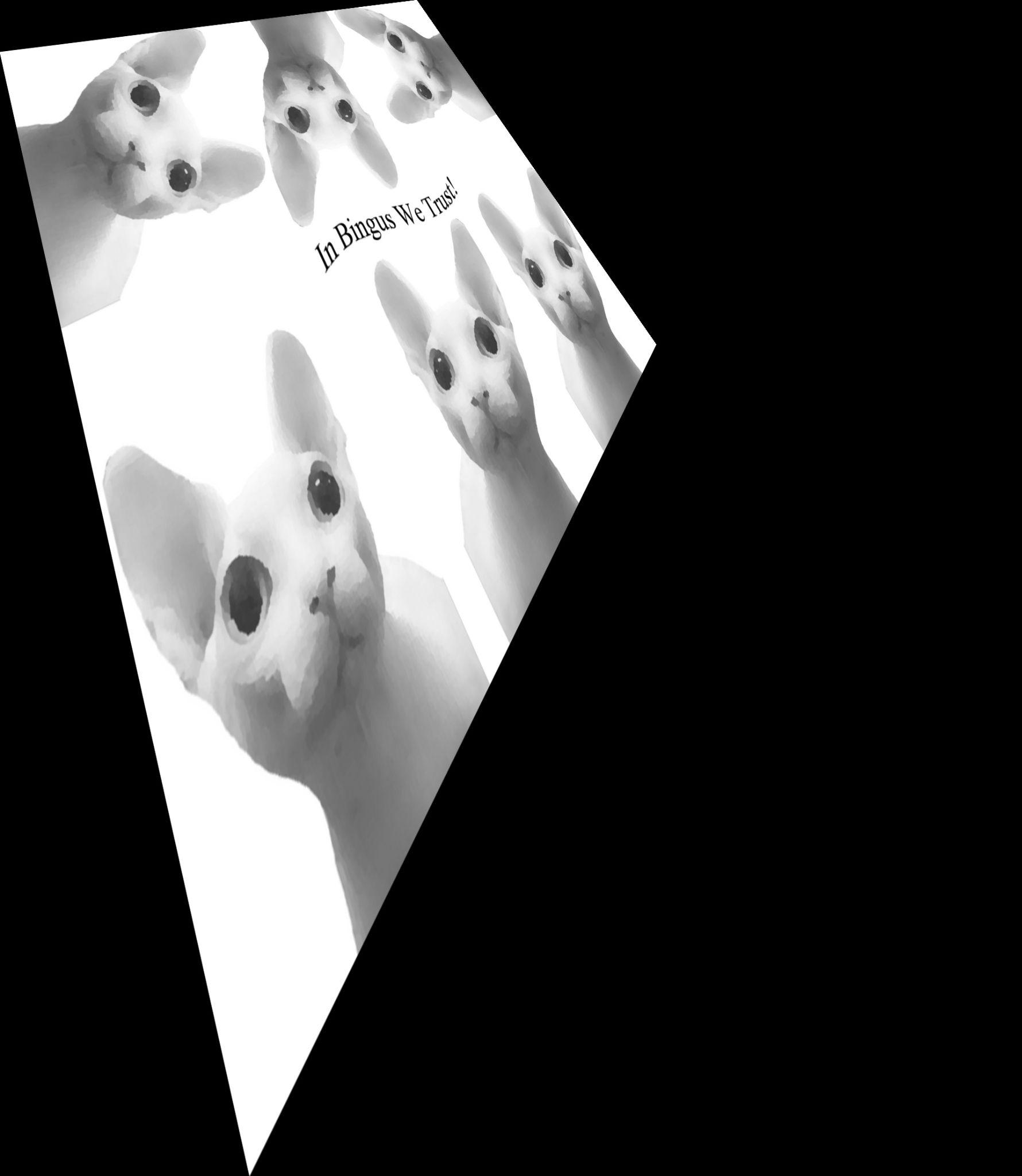
[−.1 .9 −.1]

[.0005 −.0005 1]

Here is the function calls utilized to do this task:







The next 3 images are of the transform matrices :

[29.25 13.95 20.25]

[4.95 35.55 9.45]

[0.045 0.09 45]

Here is the function calls utilized to do this task:



