Affine Transformation Project Report

Ziya Kadir TOKLUOGLU

October 30, 2024

Contents

1 Introduction

Affine transformations are fundamental operations in image processing that include scaling, rotation, shearing, and translation. These transformations allow for the manipulation of images in various ways, enabling tasks such as image alignment, augmentation, and geometric corrections. This report presents the implementation of affine transformations using forward and backward mapping techniques, with and without interpolation, and compares the results to the original image.

2 Affine Matrix

An affine transformation can be represented by a matrix that combines linear transformations such as rotation, scaling, and shearing. The general form of a 2D affine transformation matrix is:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Where:

- a and d represent scaling.
- b and c represent shearing.
- Rotation can be incorporated by setting $a = \cos(\theta)$, $b = -\sin(\theta)$, $c = \sin(\theta)$, and $d = \cos(\theta)$ for a rotation angle θ.

In this project, the following affine transformations were implemented:

1. Scaling by factor 0.5
2. Rotation by 60 degrees.
3. Scaling by a factor of 1.4 (zoom).
4. Horizontal shearing by a factor of 1.4.

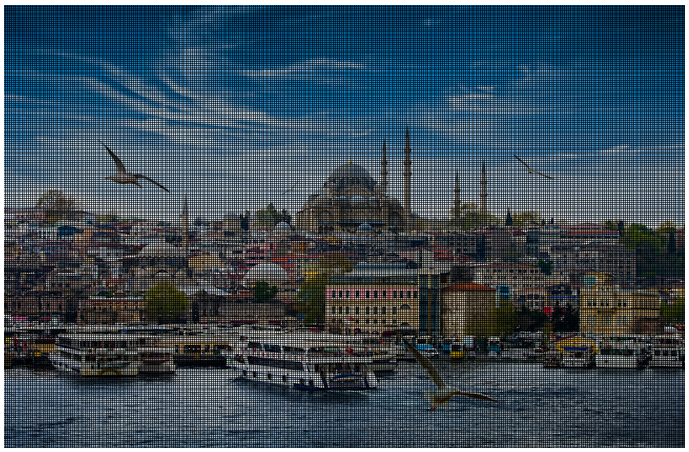
The composite affine transformation matrix is constructed by combining these individual transformations.

3 Transformation Operation Comparisons

3.1 A. Forward Mapping

Forward mapping involves mapping each pixel from the original image to its new location in the transformed image based on the affine transformation matrix. This method can sometimes result in gaps in the transformed image because not all destination pixels are guaranteed to be filled.

1.1 Scaling using forward mapping by a factor of 0.5



1.2 Scaling using forward mapping by a factor of 1.4

1.3 Rotation using forward mapping by 60 degrees

1.4 Shearing using forward mapping with a factor of 1.4

Figure 1: Forward Mapping Results

3.1.1 Comparison with Original Image

Figures 11.1, 11.2, 11.3, and 11.4 show the transformed images using forward mapping compared to the original image (Figure 2).

Figure 2: Original Image

3.1.2 Forward Mapping Results

Forward mapping is not an optimal solution when compared to the original image. If the scaling factor is greater than one, it can lead to missing pixels, as new pixels are not generated; instead, existing pixels are simply mapped to the corresponding locations in the new image. However, forward mapping can still be advantageous for matrix operations when the scaling factor is between 0 and 1.

3.2 B. Backward Mapping without Interpolation

Backward mapping maps each pixel in the transformed image back to the corresponding pixel in the original image. This approach ensures that all pixels in the transformed image are filled but may result in a blocky appearance due to the lack of interpolation.

3.1 Scaling using backward mapping without interpolation with 0.5

3.2 Scaling using backward mapping without interpolation with 1.4

3.3 Rotation using backward mapping without interpolation with degree 60

3.4 Shearing using backward mapping without interpolation with 1.4

3.2.1 Backward Mapping Results without Interpolation

Backward mapping provides an alternative approach to image scaling. Unlike forward mapping, backward mapping samples the original image to determine the pixel values for each position in the scaled image. When no interpolation is used, the nearest pixel from the original image is selected, which can lead to a blocky or pixelated appearance, especially when the scaling factor is less than one. This method ensures that no pixels are missing, but it does not smooth out the transitions between pixel values, leading to a less refined image when scaling down or up.

3.2.2 Comparison with Original Image

Figures 33.1, 33.3, and 33.4 show the transformed images using backward mapping without interpolation compared to the original image (Figure 4).

Figure 4: Original Image

3.2.3 Backward Mapping without Interpolation Results

Backward mapping without interpolation ensures that all pixels in the transformed image are populated, eliminating gaps that can occur with forward mapping. However, the absence of interpolation can lead to a blocky or pixelated appearance, especially noticeable in images with smooth gradients or detailed textures.

3.3 C. Backward Mapping with Bilinear Interpolation

Bilinear interpolation improves the quality of the transformed image by calculating pixel values based on the weighted average of the four nearest pixels in the original image. This results in a smoother and more visually appealing image.

5.1 Scaling using backward mapping with bilinear interpolation by using 0.5

5.2 Scaling using backward mapping with bilinear interpolation by using 1.4

5.3 Rotation using backward mapping with bilinear interpolation with 60 degree

5.4 Shearing using backward mapping with bilinear interpolation by using 1.4

3.3.1 Comparison with Original Image

Figures 55.2, 55.3, and 55.4 show the transformed images using backward mapping with bilinear interpolation compared to the original image (Figure 6).

Figure 6: Original Image

3.3.2 Backward Mapping Results with Bilinear Interpolation

Backward mapping with bilinear interpolation offers a more refined solution compared to the approach without interpolation. In this method, each pixel value in the scaled image is calculated by considering the four closest neighboring pixels from the original image, and their values are blended based on their distances. This technique helps in producing smoother transitions between pixels, resulting in a higher-quality scaled image. Bilinear interpolation is particularly beneficial when the scaling factor is less than one, as it minimizes pixelation and makes the image appear smoother and more continuous. However, it may introduce some blurriness when the scaling factor is greater than one, as the blending process can slightly soften sharp edges.

4 Conclusion

Affine transformations are essential tools in image processing, allowing for various manipulations such as rotation, scaling, and shearing. This project demonstrated the implementation of forward and backward mapping techniques, highlighting the advantages of using bilinear interpolation to achieve smoother and more accurate transformed images. The comparisons indicate that while forward mapping is straightforward, backward mapping with interpolation provides superior visual quality.

A Source Code

The following sections include the Python code used for implementing the affine transformations.

A.1 Main Implementation

```
def main():
    image = cv2.imread("istanbul.jpg")
    height, width, channels = image.shape
    print(f"Width: {width}, Height: {height}")

    array = con_image_to_array(image)

    angle = 60 # degrees
    theta = math.radians(angle)


    # Build rotation matrix
    # affine_matrix = [
    #     [math.cos(theta), -math.sin(theta)],
    #     [math.sin(theta), math.cos(theta)]
    # ]

    # Identity matrix
    # affine_matrix = [
    #     [1, 0],
    #     [0, 1]
    # ]

    # Scaling matrix
    # affine_matrix = [
    #     [1.5, 0],
    #     [0, 1.5]
    # ]

    # Shearing Horizental matrix
    affine_matrix = [
        [1, 1.4],
        [0, 1]
    ]

    # Shearing Vertical matrix
    # affine_matrix = [
    #     [1, 0],
    #     [0.5, 1]
    # ]


    result = affine_transformation_bilinear(array, height, width,
        channels, affine_matrix)
```

```
    result_image = con_array_to_image(result)

    cv2.imshow("Original Image", image)
    print(image.shape)


    cv2.imshow("Affine Transformed Image", result_image)
    print(result_image.shape)

    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Listing 1: Affine Transformation Implementation

A.2 Additional Functions

```
cx = width / 2
    cy = height / 2

    cos_theta = abs(transformation_matrix[0][0])
    sin_theta_1 = abs(transformation_matrix[0][1])
    sin_theta_2 = abs(transformation_matrix[1][0])

    new_width = int(width * cos_theta + height * sin_theta_1)
    new_height = int(width * sin_theta_2 + height * cos_theta)

    cx_new = new_width / 2
    cy_new = new_height / 2
```

Listing 2: calculating new dimension

We need to calculate the new dimension according to affine matrix constants and creating the new blank image

```
def con_image_to_array(image):
    height, width, channels = image.shape
    array = [[[image[y, x, c] for c in range(channels)] for x in
       range(width)] for y in range(height)]
    return array

def create_empty_array(height, width, channels):
    return [[[0 for _ in range(channels)] for _ in range(width)]
       for _ in range(height)]

def con_array_to_image(array):
    height = len(array)
    width = len(array[0])
    channels = len(array[0][0])

    image = np.zeros((height, width, channels), np.uint8)
    for y in range(height):
        for x in range(width):
```

```
            for c in range(channels):
                image[y, x, c] = array[y][x][c]

    return image
```

Listing 3: convert array and convertimage

To calculate the matrix ooperations, First we convert image to array and when all operations completed, we convert array to image

```
            x_shifted = x - cx
            y_shifted = y - cy

            dest_coords = matrix_multiplication(
                transformation_matrix, [[x_shifted], [y_shifted]])
            dest_x = dest_coords[0][0] + cx_new
            dest_y = dest_coords[1][0] + cy_new

            dest_x_int = int(round(dest_x))
            dest_y_int = int(round(dest_y))

            if 0 <= dest_x_int < new_width and 0 <= dest_y_int <
                new_height:
                for c in range(channels):
                    transformed_array[dest_y_int][dest_x_int][c]
                        = array[y][x][c]
```

Listing 4: Forward mapping

To able to perform forward mapping we iterate each pixel and do matrix calculation with using affine matrix.

```
            x_shifted = x - cx_new
            y_shifted = y - cy_new

            original_coords = matrix_multiplication(inverse, [[
                x_shifted], [y_shifted]])
            original_x = original_coords[0][0] + cx
            original_y = original_coords[1][0] + cy

            original_x_int = int(round(original_x))
            original_y_int = int(round(original_y))

            if 0 <= original_x_int < width and 0 <=
                original_y_int < height:
                for c in range(channels):
                    transformed_array[y][x][c] = array[
                        original_y_int][original_x_int][c]
```

Listing 5: Backward mapping

To able to perform backward mapping we iterate each pixel and do matrix calculation with using inverse affine matrix to able to avoid missing pixels.

```
            x_shifted = x - cx_new
```

```
            y_shifted = y - cy_new

            original_coords = matrix_multiplication(inverse, [[
                x_shifted], [y_shifted]])
            original_x = original_coords[0][0] + cx
            original_y = original_coords[1][0] + cy

            if 0 <= original_x < width - 1 and 0 <= original_y <
                height - 1:
                x0 = int(math.floor(original_x))
                x1 = x0 + 1
                y0 = int(math.floor(original_y))
                y1 = y0 + 1

                dx = original_x - x0
                dy = original_y - y0

                for c in range(channels):
                    f00 = array[y0][x0][c]
                    f10 = array[y0][x1][c]
                    f01 = array[y1][x0][c]
                    f11 = array[y1][x1][c]

                    interpolated_value = (f00 * (1 - dx) * (1 -
                        dy) +
                                          f10 * dx * (1 - dy) +
                                          f01 * (1 - dx) * dy +
                                          f11 * dx * dy)

                    transformed_array[y][x][c] = int(round(
                        interpolated_value))
```

Listing 6: Bilinear mapping

To able to perform backward mapping we iterate each pixel and do matrix calculation with using inverse affine matrix to able to avoid missing pixels. and also we look the surronded pixels to assign most accurate value

B References

https://www.youtube.com/watch?v=ORhhP–dNJA
https://www.youtube.com/watch?v=B1zCAibMcCM
https://www.youtube.com/watchv=AheaTd$_l$5$Ist = 140s$