Digital Image Processing: Homework #1

Berru Lafcı

Gebze Technical University — October 30, 2024

Introduction

This paper examines the application of affine transformation operations on an image. The code performs image scaling, rotation and shearing operations without using external libraries. This report aims to provide an in-depth understanding of how image processing techniques can be applied, supported by relevant code snippets and visual examples.

Figure 1: Original image.

Info: I did not include the zoom matrix into affine transformation because it has same matrix with the scale matrix, as I explained in the report. So I just did the transformation with scale, rotation and horizontal shear. See: 1.4.1

1 Affine Transformation Components

In this section, I will cover the affine transformation components such as scaling, rotation, shear, and zoom. I will show the matrices each component and the effects of these transformations on the image.

1.1 Scaling

Scaling is used to change the size of an image. We do this by multiplying the coordinates of each pixel by a certain scale factor. If the scale factor is greater than 1, the image becomes larger; if it is less than 1, the image becomes smaller.

Here are s_x and s_y are the scale factors on the horizontal and vertical axes respectively.

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Since the values I used are 1.4, my matrix becomes:

$$\begin{bmatrix} 1.4 & 0. & 0. \\ 0. & 1.4 & 0. \\ 0. & 0. & 1. \end{bmatrix}$$

1.2 Rotation

The rotation operation requires a transformation matrix to rotate the image around a certain angle. This matrix is calculated as follows:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Here theta represents the rotation angle. Using the value of theta to rotate by 60 degrees, my matrix becomes:

$$\begin{bmatrix} 0.5 & -0.8660254 & 0 \\ 0.8660254 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1.3 Horizontal Shear

The shearing operation creates a matrix to tilt the image along an axis. The matrix used for horizontal shearing is defined as:

$$\begin{bmatrix} 1 & s_h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Here s_h represents the shear factor. This matrix is used to shift the image along the horizontal axis. Since the value I used is 1.4, my matrix becomes:

$$\begin{bmatrix} 1 & 1.4 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1.4 Zoom

Zooming is a similar operation to scaling. It is usually used to enlarge and its matrix form is similar to the scaling matrix. My zoom matrix is as follows:

$$\begin{bmatrix} 1.4 & 0. & 0. \\ 0. & 1.4 & 0. \\ 0. & 0. & 1. \end{bmatrix}$$

1.4.1 Scale vs Zoom matrix

The scale matrix S in 3D space is defined as:

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

where s_x, s_y, and s_z are the scaling factors along the x, y, and z axes, respectively.
The zoom matrix Z can be represented as:

$$Z = \begin{bmatrix} z & 0 & 0 \\ 0 & z & 0 \\ 0 & 0 & z \end{bmatrix}$$

where $z > 1$ indicates zooming in (enlarging), and $0 < z < 1$ indicates zooming out (reducing).

In summary, zoom matrix is conceptually similar to a scale matrix because both operations modify the size of an object in a uniform manner.

2 Forward Mapping

Forward mapping involves scanning pixels from the original image and calculating their corresponding values at transformed locations in the output image. However, this process can lead to some missing pixels in the output. This occurs because multiple pixel locations from the input image may project to the same location in the output image, resulting in potential overlap where only one pixel value is represented at a given output position.

2.1 Implementation

Algorithm 1: `forward_mapping`

Input: $image$, an input image
Input: $transform\_matrix$, a transformation matrix
Result: $transformed\_image$, the transformed output image

$height, width, \_ \leftarrow image.shape$;
$transformed\_image \leftarrow$ np.zeros_like$(image)$;
$center\_x, center\_y \leftarrow \frac{width}{2}, \frac{height}{2}$;
for $y \leftarrow 0$ **to** $height - 1$ **do**
  for $x \leftarrow 0$ **to** $width - 1$ **do**
    $original\_position \leftarrow$ np.array$([x - center_x, y - center_y, 1])$;
    $new\_position \leftarrow transform\_matrix \cdot original\_position$;
    $new\_x \leftarrow$ int$(new_position[0] + center_x)$;
    $new\_y \leftarrow$ int$(new_position[1] + center_y)$;
    if $0 \leq new\_x < width \wedge 0 \leq new\_y < height$ **then**
      $transformed\_image[new\_y, new\_x] \leftarrow image[y, x]$;
    end
  end
end
return $transformed\_image$;

In this implementation of forward mapping, I utilized the center of the image as a reference point for transformations for a better view in output image.

The algorithm iterates through each pixel in the original image, adjusting the coordinates to center them around the midpoint. By applying the transformation matrix to these centered coordinates, it computes the new pixel positions in the transformed image.

After determining the new coordinates, the implementation checks if they fall within the bounds of the output image. If valid, it assigns the pixel value from the original image to the corresponding position in the transformed image.

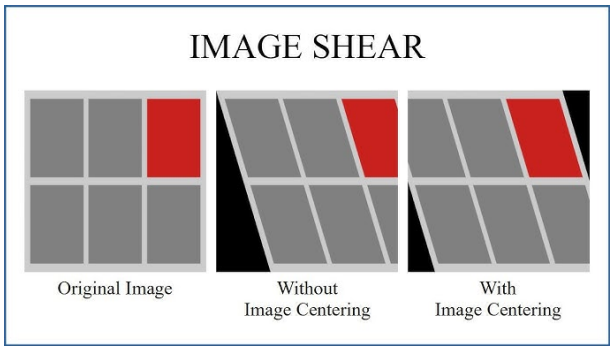

Figure 2: Sheared image.

2.2 Results

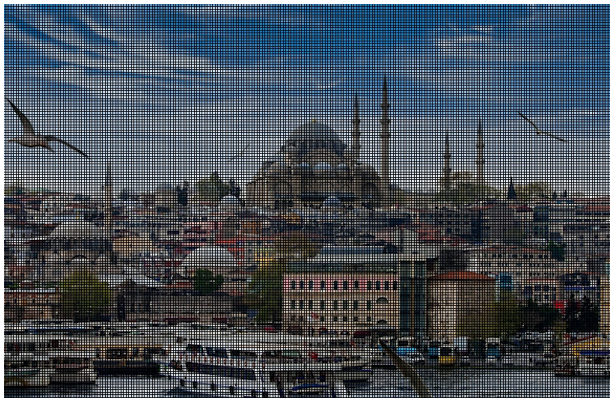

Figure 3: Forward mapping with scale matrix.

Figure 4: Forward mapping with rotation matrix.

Figure 5: Forward mapping with shear matrix.

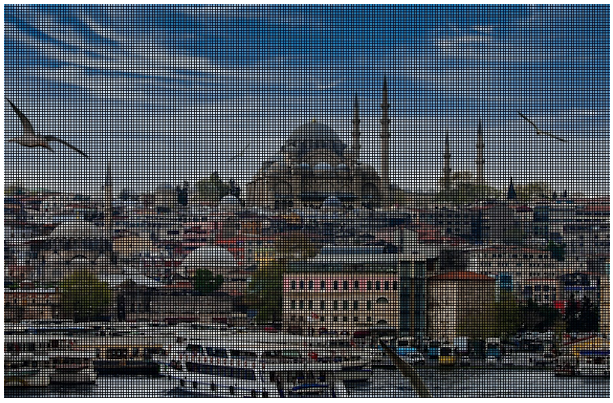

Figure 6: Forward mapping with zoom matrix.

3 Backward Mapping Without Interpolation

In backward mapping without interpolation, instead of calculating new pixel values for the transformed image using nearby pixels (which is what interpolation does), this technique looks for the nearest pixel in the original image and directly uses its color for the new image.

For each pixel in the new image, it finds the closest match in the original image and takes its color. However, this can lead to a pixelated look because it doesn't smooth out the colors between pixels like other methods do.

3.1 Implementation

Algorithm 2: `backward_mapping_no_interpolation`

Input: $image$, an input image
Input: $transform\_matrix$, a transformation matrix
Result: $transformed\_image$, the transformed output image

$height, width, \_ \leftarrow image.shape$;
$transformed\_image \leftarrow$ np.zeros_like$(image)$;
$center\_x, center\_y \leftarrow \frac{width}{2}, \frac{height}{2}$;
for $y \leftarrow 0$ **to** $height - 1$ **do**
  for $x \leftarrow 0$ **to** $width - 1$ **do**
    $original\_position \leftarrow$ np.array$([x - center_x, y - center_y, 1])$;
    $new\_position \leftarrow$ inverse_transform_matrix $\cdot\, original\_position$;
    $new\_x \leftarrow$ int$(new_position[0] + center_x)$;
    $new\_y \leftarrow$ int$(new_position[1] + center_y)$;
    if $0 \leq new\_x < width \wedge 0 \leq new\_y < height$ **then**
      $transformed\_image[y, x] \leftarrow image[new\_y, new\_x]$;
    end
  end
end
return $transformed\_image$;

The backward mapping no interpolation function calculates the center of the image and applies the inverse of the transformation matrix to determine the new pixel positions.

For each pixel in the output image, it translates the coordinates, applies the inverse transformation, and then checks if the resulting coordinates are within the bounds of the original image. If valid, it assigns the corresponding pixel value from the original image to the transformed image, effectively mapping the original image to the new space defined by the transformation.

Algorithm 3: `inverse_matrix`

Input: $matrix$, a 3x3 matrix
Result: $inverse\_matrix$, the inverse of the input matrix

if *len(matrix)* $== 3 \wedge$ *len(matrix[0])* $== 3$ **then**
  $a, b, c \leftarrow matrix[0]$;
  $d, e, f \leftarrow matrix[1]$;
  $g, h, i \leftarrow matrix[2]$;
  $determinant \leftarrow a \cdot e \cdot i + b \cdot f \cdot g + c \cdot d \cdot h - c \cdot e \cdot g - b \cdot d \cdot i - a \cdot f \cdot h$;
  $inverse\_matrix \leftarrow \begin{bmatrix} \frac{e \cdot i - f \cdot h}{determinant} & \frac{c \cdot h - b \cdot i}{determinant} & \frac{b \cdot f - c \cdot e}{determinant} \\ \frac{f \cdot g - d \cdot i}{determinant} & \frac{a \cdot i - c \cdot g}{determinant} & \frac{c \cdot d - a \cdot f}{determinant} \\ \frac{d \cdot h - e \cdot g}{determinant} & \frac{g \cdot b - a \cdot h}{determinant} & \frac{a \cdot e - b \cdot d}{determinant} \end{bmatrix}$;
  return $inverse\_matrix$;
end
else
  return *ValueError("Matrix is not 3x3")* ;
end

In the context of backward mapping, we utilize the inverse matrix to determine where each pixel in the output image should come from in the input image.

For a 3×3 matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

the inverse A^{-1} can be computed using the formula:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ae & ae - bd \end{bmatrix}$$

where the determinant $\det(A)$ is given by:

$$\det(A) = aei + bfg + cdh - ceg - bdi - afh$$

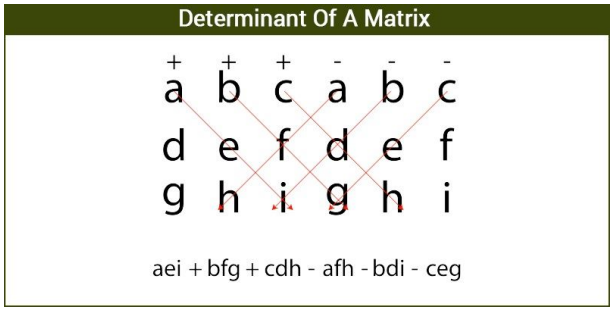

Figure 7: Determinant of a matrix visually.

3.2 Results

Figure 8: Backward mapping without interpolation with scale matrix.

Figure 9: Backward mapping without interpolation with rotation matrix.

Figure 10: Backward mapping without interpolation with shear matrix.

Figure 11: Backward mapping without interpolation with zoom matrix.

4 Backward Mapping with Bilinear Interpolation

In this method, each pixel in the destination image is mapped back to a corresponding position in the source image, similar to backward mapping without interpolation. However, instead of simply copying the color from the nearest pixel, bilinear interpolation calculates the color based on a weighted average of the colors of the four nearest pixels in the original image. These four pixels form a rectangle around the mapped position. By considering the relative distances of the new pixel from each of these four pixels, the algorithm determines a more precise color value for the new pixel. This results in smoother transitions and reduced pixelation, making the transformed image appear more natural and visually appealing.

4.1 Implementation

Algorithm 4: `bilinear_interpolation`

Input: $image$, an input image
Input: x, x-coordinate for interpolation
Input: y, y-coordinate for interpolation
Result: $pixel\_value$, the interpolated pixel value

if $x < 0 \vee y < 0 \vee x \geq image.shape[1] - 1 \vee y \geq image.shape[0] - 1$ **then**
  return $[0, 0, 0]$;
end
$x0 \leftarrow \text{int}(x)$;
$x1 \leftarrow \min(x0 + 1, image.shape[1] - 1)$;
$y0 \leftarrow \text{int}(y)$;
$y1 \leftarrow \min(y0 + 1, image.shape[0] - 1)$;
$a \leftarrow x - x0$;
$b \leftarrow y - y0$;
$top\_left \leftarrow image[y0, x0]$;
$top\_right \leftarrow image[y0, x1]$;
$bottom\_left \leftarrow image[y1, x0]$;
$bottom\_right \leftarrow image[y1, x1]$;
$pixel\_value \leftarrow (1 - a) \cdot (1 - b) \cdot top\_left + a \cdot (1 - b) \cdot top\_right + (1 - a) \cdot b \cdot bottom\_left + a \cdot b \cdot bottom\_right$;
return $pixel\_value.astype(np.uint8)$;

The `bilinear_interpolation` function calculates the integer coordinates of the surrounding pixels (x_0, x_1, y_0, y_1) and determines the fractional distances a and b from these integers. The function then retrieves the pixel values at the corners of the surrounding pixels and uses these values to compute the interpolated pixel value based on a weighted average. This weighted average takes into account the distances a and b, effectively blending the contributions of the corner pixels to produce a smooth transition for the pixel value at the given coordinates.

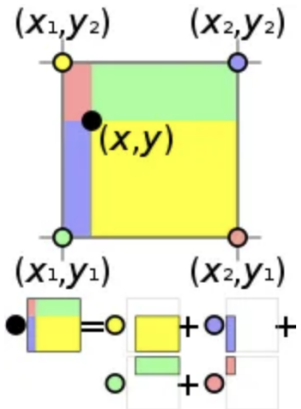

Figure 12: Bilinear interpolation visually.

Algorithm 5: `backward_mapping_with_interpolation`

Input: $image$, an input image
Input: $transform\_matrix$, a transformation matrix
Result: $transformed\_image$, the transformed output image

$height, width, \_ \leftarrow image.shape$;
$transformed\_image \leftarrow$ np.zeros_like$(image)$;
$center\_x, center\_y \leftarrow \frac{width}{2}, \frac{height}{2}$;
$inverse\_transform\_matrix \leftarrow$ inverse_matrix$(transform_matrix)$;
for $y \leftarrow 0$ **to** $height - 1$ **do**
  for $x \leftarrow 0$ **to** $width - 1$ **do**
    $original\_position \leftarrow$ np.array$([x - center_x, y - center_y, 1])$;
    $new\_position \leftarrow inverse\_transform\_matrix \cdot original\_position$;
    $new\_x \leftarrow new\_position[0] + center\_x$;
    $new\_y \leftarrow new\_position[1] + center\_y$;
    $pixel\_value \leftarrow$ bilinear_interpolation$(image, new_x, new_y)$;
    $transformed\_image[y, x] \leftarrow pixel\_value$;
  end
end
return $transformed\_image$;

This function is similar to `backward_mapping_no_interpolation` except that it incorporates bilinear interpolation to provide smoother pixel values in the transformed image.

For each pixel, it finds the original position and applies bilinear interpolation, which estimates the pixel value based on its neighboring pixels. This addition of bilinear interpolation enhances the quality of the output image by blending the pixel values, unlike the previous function, which simply maps pixels without considering their surrounding context.

4.2 Results

Figure 13: Backward mapping with interpolation with scale matrix.

Figure 14: Backward mapping with interpolation with rotation matrix.

Figure 15: Backward mapping with interpolation with shear matrix.

Figure 16: Backward mapping with interpolation with zoom matrix.

5 Final Comparison of Affine Transformation Results

The affine transformation was achieved by applying all three transformations: scaling, rotation, horizontal shear. The resulting images are compared across the three methods: forward mapping, backward mapping without interpolation, and backward mapping with bilinear interpolation.

5.1 Transformation Matrix Calculation

$$T = S \cdot R \cdot H$$

where:

- S is the scaling matrix,
- R is the rotation matrix, and
- H is the shearing matrix.

The final transformation matrix T is computed by multiplying these matrices in the order of their application (scaling, then rotation, and finally shearing). The resulting matrix T contains the combined effect of all transformations applied to the points in space.

5.2 Implementation

To combine the transformations, the following operations were performed:

$$\texttt{transformation\_matrix} = \texttt{matrix\_multiply}(\texttt{scale\_matrix}, \texttt{rotation\_matrix})$$

$$\texttt{transformation\_matrix} = \texttt{matrix\_multiply}(\texttt{transformation\_matrix}, \texttt{shear\_matrix})$$

Algorithm 6: `matrix_multiply`

Input: A, the first matrix
Input: B, the second matrix
Result: $result$, the product of the matrices

$rows\_A \leftarrow \text{len}(A)$;
$cols\_A \leftarrow \text{len}(A[0])$;
$rows\_B \leftarrow \text{len}(B)$;
$cols\_B \leftarrow \text{len}(B[0])$;
if $cols\_A \neq rows\_B$ **then**
  return *ValueError("not compatible matrices")* ;
end
$result \leftarrow [[0 \text{ for } \text{ in range}(cols\_B)] \text{ for } \text{ in range}(rows\_A)]$;
for $i \leftarrow 0$ **to** $rows\_A - 1$ **do**
  for $j \leftarrow 0$ **to** $cols\_B - 1$ **do**
    for $k \leftarrow 0$ **to** $cols\_A - 1$ **do**
      $result[i][j] \leftarrow result[i][j] + A[i][k] \cdot B[k][j]$;
    end
  end
end
return $result$;

The `matrix_multiply` function performs matrix multiplication between two compatible matrices A and B. The function initializes a result matrix with zeros and calculates the product by iterating through the rows of A and the columns of B, summing the products of corresponding elements.

5.3 Results

Forward Mapping

Forward mapping shows gaps due to unfilled pixel locations. This method struggles to maintain continuity, leading to visible artifacts in the transformed image.

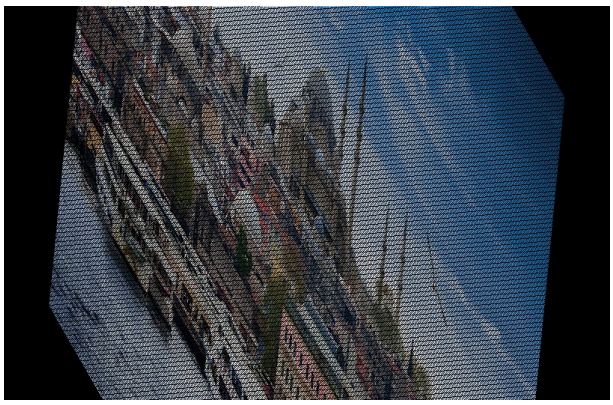

Figure 17: Forward mapping with transformation matrix.

Figure 18: Zoomed picture to see the gaps.

Backward Mapping (No Interpolation)

Backward mapping without interpolation results in a pixelated image with rough edges. The absence of interpolation causes the transformation to lose smoothness, making the output less visually appealing.

Figure 19: Backward mapping without interpolation with transformation matrix.

Figure 20: Zoomed picture to see the pixels.

Backward Mapping (Bilinear Interpolation)

Backward mapping with bilinear interpolation produces the smoothest result, preserving image quality and minimizing artifacts. This method effectively blends pixel values, leading to a more coherent and aesthetically pleasing image after transformation.

Figure 21: Backward mapping without interpolation with transformation matrix.

Figure 22: Zoomed picture to see the smoothness.

6 Conclusion

In conclusion, I learned how to apply affine transformations, including scaling, rotation, zooming, and shearing, to images using different mapping techniques. I observed that the results varied significantly between methods—forward mapping, backward mapping without interpolation, and backward mapping with bilinear interpolation. Finally, I found that bilinear interpolation produced the best visual results, preserving details and minimizing artifacts in the transformed images.