Lab 4: Image Registration

Daniel Sherman (0954083), *Student, University of Guelph*

***Abstract*—Image scaling, rotating, and translating were explored on a set of input coordinates and output coordinates, to determine an affine transformation matrix relating the two sets of coordinates. MATLAB routines to perform image scaling, rotating, and translating were written. Unknown transformations were applied to an image, and through optimization, the unknown set of transformations were found.**

***Index Terms*—Image Registration, Transformation Matrix, Affine, Scale, Rotation, Translation**

I. Introduction

UNDERSTANDING where features are in images has great value in numerous applications. Scaling, rotating, and translating an image can allow one to study the content of the image in a more robust way than just studying the image by inspection. Image registration is the process where one image is mapped to known coordinates, or another image.

Registration can be used in a situation where multiple images are taken, for example multiple x-rays at slightly different angles, zoom, or positions. Registration can take the different images and shift them to be in the same position for analysis. Furthermore, registration can be used to extract information from the same image being taken from multiple imaging modalities, such as an x-ray and MRI of the same anatomy.

In this lab, an affine transformation matrix was found given two sets of coordinates. Image transformation was explored. Both experiments were used to determine a set of unknown transformations on an image.

II. Methods

'mri.jpg' was used and loaded into MATLAB to serve as the reference image, and is seen in Figure 1.

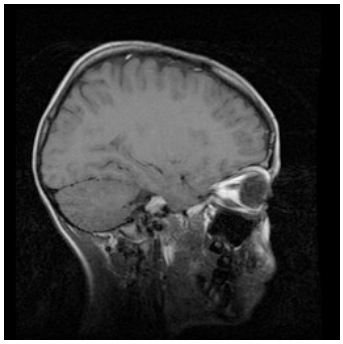


Fig. 1: 'mri.jpg' (Size 256x256 pixels)

A version of Figure 1 was sent, which had unknown transformations applied to it was also loaded into MATLAB. The sent image is seen in Figure 2.

Lab Report received May 26, 2020

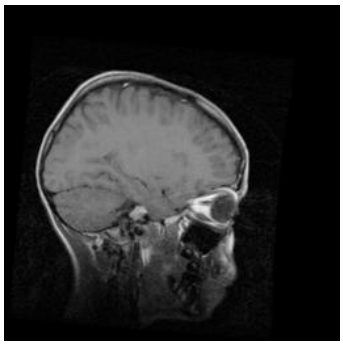


Fig. 2: Mystery image with unknown rotation, translation, and scaling (Size 256x256 pixels)

Two sets of coordinates were also sent, one serving as the reference, and the other which had unknown scaling, rotation, and translation applied to them.

A MATLAB function was written to accept the two sets of coordinates and compute the affine transformation matrix in a homogeneous coordinate system, in the form seen in Equation 1.

$$\begin{bmatrix} x_1' \\ y_1' \\ 1 \end{bmatrix} = \begin{bmatrix} S\cos\theta\sin\theta & \sin\theta & t_x \\ -\sin\theta & S\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \quad (1)$$

Where S is a scaling factor, θ is the rotation angle, about the midpoint of the image, t_x, t_y are translations in the x and y directions, x_1, y_1 are the input points, and x_1', y_1' are the transformed output points.

As the two vectors in Equation 1 were given in the form of the two sets of coordinates, the affine transformation matrix is able to be found. To test the validity of the found transformations, the matrix was multiplied with the input points, and the average Euclidean distance was calculated as a validity metric.

The translations were found by picking off the matrix values from Equation 1. The rotation angle and scaling were found by solving the nonlinear system of equations given by the (1,1), and (1,2) elements in the affine transformation matrix.

MATLAB functions to perform image rotation, translation, and scaling with Bilinear Interpolation were written, and the transformations found in Equation 1 were applied to Figure 1.

MATLAB's `fminsearch()` command was utilized to determine the transformations that were applied to Figure 2. To do so, a function that calculates the mean squared error (MSE) between the mystery image and a transformed version of Figure 1 was calculated. When `fminsearch()` was minimized, it was determined that the two images were near identical, and the set of transformations to get that image were

the transformations applied to obtain the mystery image in the first place.

MATLAB code used in this lab can be seen in section B.

III. Results and Discussion

The transformation that relates the given input points and the given output points is seen in Equation 2.

$$A = \begin{bmatrix} 1.0307 & -0.1555 & 12.4665 \\ 0.1567 & 1.0383 & 9.5884 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

Relating Equation 2 to the matrix in Equation 1, $\theta = 0.156\,08\,\text{rad}$, $T_x = 12.4665$, $T_y = 9.5884$, and $S = 1.0434$.

After finding the transformation matrix relating the two sets of coordinates, the transformation matrix was applied to the input coordinates to verify the results. Both the given output coordinates and the calculated output coordinates are seen in Figure 3. The average Euclidean distance between the given output points and the calculated output points was 0.3429 pixels, indicating that the calculated transformation matrix was very accurate.

The transformation in Equation 2 was applied to Figure 1, and is seen in Figure 4. Functions were written to apply the rotation, scaling, and translation using Bilinear Interpolation.

To determine the set of unknown transformations to transform Figure 1 into Figure 2, the similarity measure of Mean Squared Error (MSE) was defined. Minimizing the MSE between the two images is indicative that they are the same. Therefore, the set of rotation, scaling, and translation that when applied to Figure 1, minimizes the MSE are in fact the transformations that were applied to get the mystery image. The MSE between the transformed original image and the mystery image decreases over iterations of `fminsearch()`, which is indicative of the transformations being closer and closer to the mystery image. The final MSE between the two images was 10.7959 (Figure 5), which is indicative of a good initial guess, and final output. The optimized transformations were $\theta = -0.0843\,\text{rad}$, $S_{x,y} = 0.8564$, $T_x = -29.9966$, and $T_y = -3.6621$. The mystery image and the transformed original image with the above parameters are seen side by side in Figure 6. The two images look near identical.

IV. Conclusion

Sets of points, and an image were supplied that had unknown transformations applied to them. MATLAB routines were written to determine the unknown set of transformations to great success. For the mystery set of points, only matrix algebra and nonlinear equation solving was necessary to find the set of transformations to relate the input and output set of points. The small Euclidean distance between the given output points and the calculated output points is indicative of high accuracy of the transformations.

The found transformations were applied to a sample image, along with Bilinear Interpolation in MATLAB.

Minimizing the MSE between a mystery image and a transformed reference image proved to be an effective way to determine the set of transformations that related the mystery image to the reference image.

Appendix A
Image Results

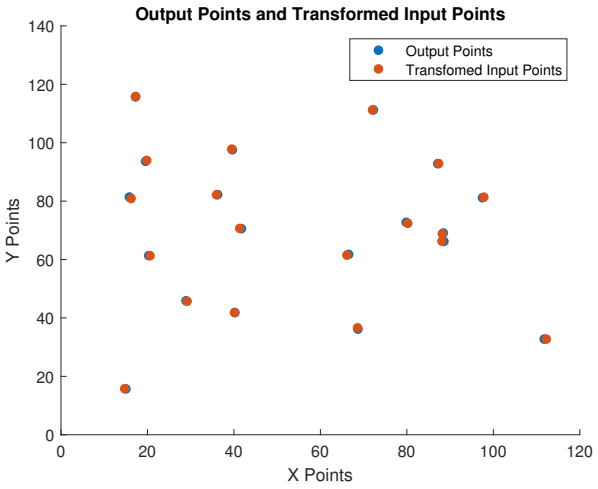
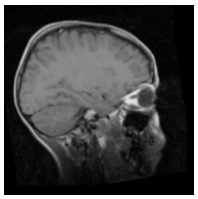


Fig. 3: Given output coordinates and calculated output coordinates from Equation 1 and the found transformation parameters



θ = 0.15608, T_x = 12.4665, T_y = 9.5884, S_x = 1.0434, S_y = 1.0434

Fig. 4: Figure 1 after being transformed by the listed parameters

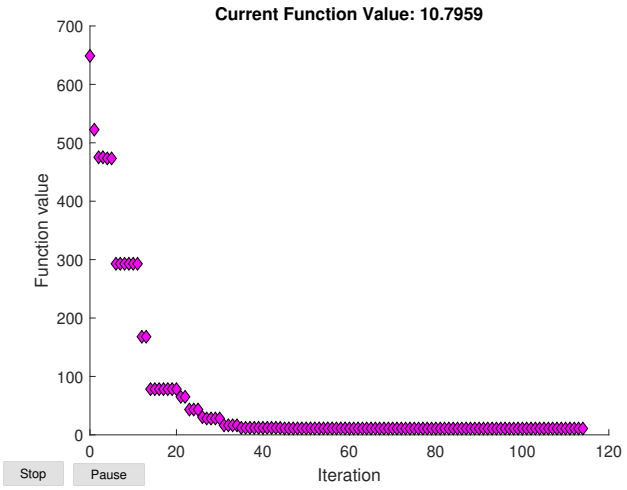


Fig. 5: MSE between the mystery image and the guessed transformation over iterations

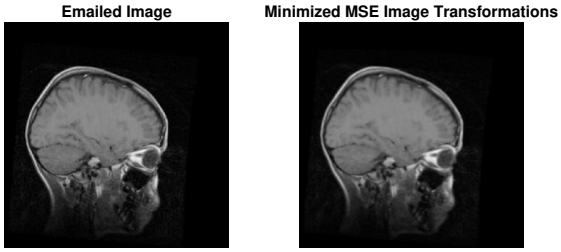


Fig. 6: Mystery Image (left) and the optimized transformations applied to Figure 1 (right)

APPENDIX B
MATLAB CODE

Listing 1: Main code used in lab

```
%% ENGG 4660: MEDICAL IMAGE PROCESSING
% LAB 4: IMAGE REGISTRATION
% DANIEL SHERMAN
% 0954083
% MARCH 7, 2020

%% START OF CODE

close all
clear all
clc

%% LOAD IN FILES

email = imread('img_xfm07.jpg');
points = load('points07.txt');
mri = imread('mri.jpg');

%points in the reference image
x1 = points(1,:);
y1 = points(2,:);
%points in the given image
x2 = points(3,:);
y2 = points(4,:);

disp('Done loading files')

%% FIND TRANSFORMATION MATRIX, AND TRANSFORMATIONS

[tx_matrix, angle, scale, tx, ty] = affine_tx(x1, y1, x2, y2); %find the transformation matrix

angle = double(angle); %convert to usable type
scale = double(scale);

%apply transformation found on 'mri.jpg'
transform_image_new(mri, angle(2), tx, ty, scale(2), scale(2))

disp('Done finding the affine transformation matrix')

%% FIND THE MYSTERY IMAGE TRANSFORMATION

%display optimization function output (MSE) over iteration number
options = optimset('PlotFcns',@optimplotfval);

%determine optimal transformation values by calling the function
%mean_sq_err()
optimized = fminsearch(@mean_sq_err, [deg2rad(-7), 0.8565, -25, -3], options)

%scale original image with optimized scale
optimized_scale = bilinear_interp_scale(mri, optimized(2), optimized(2));
%rotate original image with optimized angle
optimized_rot = bilinear_interp_angle(optimized_scale, optimized(1));
%translate original image with optimized translation
optimized_img = bilinear_interp_translate(optimized_rot, optimized(3), optimized(4));

%display mystery image and optimized transformations applied to original
%image
figure()
subplot(1,2,1)
imshow(email)
title('Emailed Image')
subplot(1,2,2)
imshow(uint8(optimized_img))
title('Minimized MSE Image Transformations')
```

Listing 2: Function that accepts two sets of points to calculate the Affine Transformation Matrix and calculates the transformation parameters and applies the transformation to the points and calculates the average Euclidean distance between the given output and the calculated output

```
function [tx_matrix, out_theta, out_scale, t_x, t_y] = affine_tx(x1, y1, x2, y2)
%% DOCUMENTATION

% FUNCTION COMPUTES THE AFFINE TRANSFORM BETWEEN TWO SETS OF POINTS,
% IN THIS CASE, x1, y1, x2, y2
% WHERE x1, y1 ARE POINTS IN THE REFERENCE IMAGE,
% AND x2, y2 ARE POINTS IN THE TRANSFORMED IMAGE

% MADE BY: DANIEL SHERMAN
% MARCH 7, 2020

%% START OF CODE

%% CREATE THE AFFINE TRANSFORMATION MATRIX

img_input_v = [x1;y1;ones(size(x1))]; %create points in original image vector
img_output_x = [x2;y2;ones(size(x2))]; %create points in transformed image vector

%matrix multiplication to get the 3x3 transformation matrix
tx_matrix = (img_output_x*(img_input_v.'))*inv(img_input_v*img_input_v.');

%% EVALUATE THE TRANSFORMATION MATRIX

%apply the transformation matrix to the given set of input points
affine_output = tx_matrix*img_input_v;

x_output = affine_output(1,:); %parse out x coordinates
y_output = affine_output(2,:); %parse out y coordinates

figure()
scatter(x_output,y_output, 'filled') %plot the output of the transformation matrix
hold on
scatter(x2,y2, 'filled') %plot the given output points
legend('Output Points', 'Transfomed Input Points')
xlabel('X Points')
ylabel('Y Points')
title('Output Points and Transformed Input Points')

%euclidean distance between given output points and calculated transformed points
err = sqrt((x_output - x2).^2 + (y_output - y2).^2);
av_err = mean(err); %average euclidean error

%% FIND THETA, TX, TY, AND SCALE

t_x = tx_matrix(1,3); %grab x translation from matrix element
t_y = tx_matrix(2,3); %grab y translation from matrix element

syms theta scale %symbolically define the angle and scaling factor
%define equations based on the transformation matrix elements and
%theoretical definitions
eq1 = scale*cos(theta) == tx_matrix(1,1);
eq2 = -sin(theta) == tx_matrix(1,2);

[out_theta, out_scale] = solve([eq1, eq2], [theta, scale]); %solve nonlinear system of equations
```

Listing 3: Function that applies image rotation and Bilinear Interpolation

```
function out_img = bilinear_interp_angle(in_img, theta)
%% DOCUMENTAION

% FUNCTION ACCEPTS AN IMAGE AND PERFORMS BILINEAR INTERPOLATION FOR A
% ROTATED IMAGE ONLY

% MADE BY: DANIEL SHERMAN
% MARCH 9, 2020
```

```
%% START OF CODE

[m,n] = size(in_img); %find size of image
X_m = m/2; %find midpoint of rows
Y_m = n/2; %find midpoint of columns

J = zeros(m, n); %initialize new image

for x_i = [1:m]
    for y_i = [1:n]
        %rotate about the centre of the image
        x2 = cos(theta)*(x_i - X_m) + sin(theta)*(y_i - Y_m) + X_m;
        y2 = -sin(theta)*(x_i - X_m) + cos(theta)*(y_i - Y_m) + Y_m;

        r = floor(x2);
        c = floor(y2);

        d_r = x2 - r;
        d_c = y2 - c;

        %run bilinear interpolation only where pixel values exist
        if r <= m - 1 && c <= n - 1 && r >=2 && c >= 2
            J(x_i, y_i) = in_img(r,c)*(1 - d_r)*(1 - d_c) + ...
                in_img(r + 1, c)*(d_r)*(1 - d_c) + ...
                in_img(r, c + 1)*(d_c)*(1 - d_r) + ...
                in_img(r + 1, c + 1)*(d_r)*(d_c); %bilinear interpolation formula
        else
        end

    end
end

out_img = J;
```

Listing 4: Function that applies image scaling and Bilinear Interpolation

```
function out_img = bilinear_interp_scale(in_img, Sr, Sc)
%% DOCUMENTAION

% FUNCTION ACCEPTS AN IMAGE AND PERFORMS BILINEAR INTERPOLATION FOR A
% SCALED IMAGE ONLY

% MADE BY: DANIEL SHERMAN
% MARCH 9, 2020

%% START OF CODE

[m,n] = size(in_img); %find size of input image

R_p = round(m*Sr); %find new number of rows
C_p = round(n*Sc); %find new number of columns

J = zeros(R_p, C_p); %initialize new image

%iterate through pixel values to determine new pixel values
for r_p = [1:R_p]
    for c_p = [1:C_p]
        r_f = r_p/Sr;
        c_f = c_p/Sc;

        r = floor(r_f); %new number of rows
        c = floor(c_f); %new number of columns

        d_c = c_f - c; %difference between current column and new number of columns
        d_r = r_f - r; %difference between current row and new number of rows

        %run bilinear interpolation only where pixel values exist
        if r <= m - 1 && c <= n - 1 && r >=2 && c >= 2
            J(r_p, c_p) = in_img(r,c)*(1 - d_r)*(1 - d_c) + ...
```

```
            in_img(r + 1, c)*(d_r)*(1 - d_c) + ...
            in_img(r, c + 1)*(d_c)*(1 - d_r) + ...
            in_img(r + 1, c + 1)*(d_r)*(d_c); %bilinear interpolation formula
        else
        end

    end
end

[m_j, n_j] = size(J); %collect size of scaled image

if m_j <= m; %check if new image is smaller than original
    out_img = J;
    out_img(m_j + 1:m, n_j:n) = 0; %ensure same size by padding with 0 to the original dimension
else m_j > m; %check if new image is larger than original
        out_img = J(1:m, 1:n); %take subsection of new image to maintain same size
end
```

Listing 5: Function that applies image translation and Bilinear Interpolation

```
function out_img = bilinear_interp_translate(in_img, tx, ty)
%% DOCUMENTAION

% FUNCTION ACCEPTS AN IMAGE AND PERFORMS BILINEAR INTERPOLATION FOR A
% TRANSLATED IMAGE ONLY

% MADE BY: DANIEL SHERMAN
% MARCH 9, 2020

%% START OF CODE

[m,n] = size(in_img);

J = zeros(m, n); %initialize new image

for x_i = [1:m]
    for y_i = [1:n]

        %translate image pixels
        x2 = x_i + tx;
        y2 = y_i + ty;

        r = floor(x2);
        c = floor(y2);

        d_r = x2 - r;
        d_c = y2 - c;

        %run bilinear interpolation only where pixel values exist
        if r <= m - 1 && c <= n - 1 && r >=2 && c >= 2
            J(x_i, y_i) = in_img(r,c)*(1 - d_r)*(1 - d_c) + ...
                in_img(r + 1, c)*(d_r)*(1 - d_c) + ...
                in_img(r, c + 1)*(d_c)*(1 - d_r) + ...
                in_img(r + 1, c + 1)*(d_r)*(d_c); %bilinear interpolation formula
        else
        end

    end
end

out_img = J;
```

Listing 6: Function that transforms an image given transformation parameters

```
function output_img = transform_image_new(in_img, theta, tx, ty, Sx, Sy)
%% DOCUMENTATION

% FUNCTION ACCEPTS AN IMAGE, ANGLE, TRANSLATION IN X AND Y, AND SCALING FACTOR IN X AND Y
% FUNCTION RETURNS A TRANSLATED IMAGE BY THE FACTORS SPECIFIED
```

```
% MADE BY: DANIEL SHERMAN
% MARCH 9, 2020

%% START OF CODE

scale = bilinear_interp_scale(in_img, Sx, Sy); %apply scaling

rotate = bilinear_interp_angle(scale, theta); %apply rotation

translate = bilinear_interp_translate(rotate, tx, ty); %apply translation

%% DISPLAY IMAGE

figure()
imshow(uint8(translate))
xlabel(strcat(['\theta = ', num2str(theta), ', T_x = ', ...
    num2str(tx), ', T_y = ', num2str(ty), ', S_x = ' num2str(Sx), ', S_y = ', num2str(Sy)]))
```

Listing 7: Function that applies given transformation parameters and calculates the MSE between the transformed image and the mystery image

```
function m_sq_err = mean_sq_err(guess)
%% DOCUMENTATION

% FUNCTION ACCEPTS 2 IMAGES (ONE TRANSFORMED FROM THE EMAIL, ONE NOT) AND
% TRANSFORMATIONS (THETA, SX, SY, TX, TY)
% FUNCTION PERFORMES THE TRANSFORMATION ON THE REFERENCE IMAGE,
% AND CALCULATES THE MEAN SQUARED ERROR BETWEEN THE TWO
% FUNCTION IS USED TO FIND THE TRANSFORMATION OF THE MYSTERY IMAGE

% MADE BY: DANIEL SHERMAN
% MARCH 20, 2020

%% START OF CODE

%parse out initial guess from input vector
theta = guess(1);
Sx = guess(2);
tx = guess(3);
ty = guess(4);

%load in original image (hardcoded)
ref_img = imread('mri.jpg');

[m,n] = size(ref_img); %grab size of original image

% ~~~ UNCOMMENT THE NEXT 3 LINES AND COMMENT THE LINE THAT DEFINES
% 'mys_img' TO RUN THIS FUNCTION WITH AN IMAGE OF ARBITRARY TRANSFORMATIONS ~~~
% NEXT THREE LINES WERE USED FOR FUNCTION VERIFICATION
% mys_shift = bilinear_interp_translate(ref_img, 10, 10);
% mys_rot = bilinear_interp_angle(mys_shift, deg2rad(33));
% mys_img = bilinear_interp_scale(mys_rot, 1.33, 1.33);

%load in mystery image
mys_img = imread('img_xfm07.jpg');

%% TRANSFORM REFERENCE IMAGE BASED ON GIVEN TRANSFORMATIONS

scale = bilinear_interp_scale(ref_img, Sx, Sx); %scale original image with guess
rotate = bilinear_interp_angle(scale, theta); %rotate origianl image with guess
full_tfd_img = bilinear_interp_translate(rotate, tx, ty); %translate original image with guess

%% FIND MEAN SQUARED ERROR BETWEEN THE TRANSFORMED IMAGE AND THE MYSTERY IMAGE

m_sq_err = immse(double(full_tfd_img), double(mys_img));
```