Signal and image processing: Assignment 6

Mariuta Nicolae 02-10-2014

Exercise 1.1.

In case of translation by one pixel to the right, then:

$$\tilde{I}(x,y) = I(x-1,y)$$

where$\tilde{I}$ is the translated image and I is the original image.

This transformation can be obtained by using a filter: $F = (1 \quad 0 \quad 0)$ which takes the value from pixel to the left and moves it to the right pixel.

```
%1.1
filter = [1,0,0];
I = imread('lena.tiff');
It = imfilter(I,filter);

subplot(1,2,1);imshow(I);
subplot(1,2,2);imshow(It);
```

Fig.1 Code for translating the picture with one pixel to the right.

I have used the lena image and I have applied the filter $F = (1 \quad 0 \quad 0)$ to the image using Matlab function *imfilter*.

The filter F could also be the 3x3 matrix $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ but will have the same effect.

By running the code from above, the translation is visible by that there is a column with 0 pixels on the left of the image. That happens because the image got out of the boundaries, the right column has disappeared and the left one was replaced by column of black pixels.

Fig.2 The result for translating image to the right by one pixel

Exercise 1.2.

At the previous exercise, I have managed to translate the image to the right by one pixel by using the filter$F = (1 \quad 0 \quad 0)$. That means the image can be translated n pixels to the right by

applying that filter n times and the image can be translated n pixels to the left by applying the filter $F = (0 \quad 0 \quad 1)$ n times.

Of course, the image can also be translated up or down by using the filters $F = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ for translating the image down and the filter $F = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ for translating up.

I have created a function for translating the image easier.

```
function T = translate( I, x, y )

T = I;

for i = 1 : abs(x)
    if (x < 0)
        F = [0,0,1];
        T = imfilter(T,F);
    end
    if (x > 0)
        F = [1,0,0];
        T = imfilter(T,F);
    end

end

for i = 1 : abs(y)
    if (y < 0)
        F = [1;0;0];
        T = imfilter(T,F);
    end
    if (y > 0)
        F = [0;0;1];
        T = imfilter(T,F);
    end

end
```

Fig.3 Function for translating the image for translating the image

Parameter I is the input image and x is number of pixels to translate the image horizontally. If x is positive, the image is translated to the right and if x is negative the image is translated to the left. Parameter y is number of pixels to translate the image vertically. If y is positive, the image is translated up and if y is negative, the image is translated down.

For the implementation, I have just applied the filters from above the number of times that is needed. Again, I have used the Matlab function *imfilter* to apply the filters because it works very fast.

For the testing, I created the image with white square in the middle and applied the filter.

```
%1.2
x = 101;
y = 101;
sql = 31;
sqw = 31;

I = zeros(x,y);

for r = (x/2)-(sql/2) : (x/2)+(sql/2)
    for c = (y/2)-(sqw/2) : (y/2)+(sqw/2)
        I(r,c) = 1;
    end
end

It = translate(I,30,30);

subplot(1,2,1);imshow(I);
subplot(1,2,2);imshow(It);
```

Fig.4 Code for creating image with white square in the middle and apply the translation function that was created.

I have created image with odd sizes and placed the white square in the middle. I have tried to create the image so to be easy to change dimensions by changing only few parameters. I have also applied the *translate* function that I created to see if it works.

Just as expected, because x is positive value, the square is translated to the right and the image is also translated down because y is negative number.

Fig.5 Translating the white square by x = 30, y = -30

Fig.6 Translating the white square by x = 50, y = -30

By applying translation it happens that the image goes out of boundaries. In the case with white square in middle of black image, the effect can be seen only if values for translating are high enough so the square goes out of the image. Function *imfilter* has few boundary conditions. In case no other option is used, the parts of the image that go out of the matrix are gone and the areas that are no longer covered by the image are filled with '*0*' pixels.

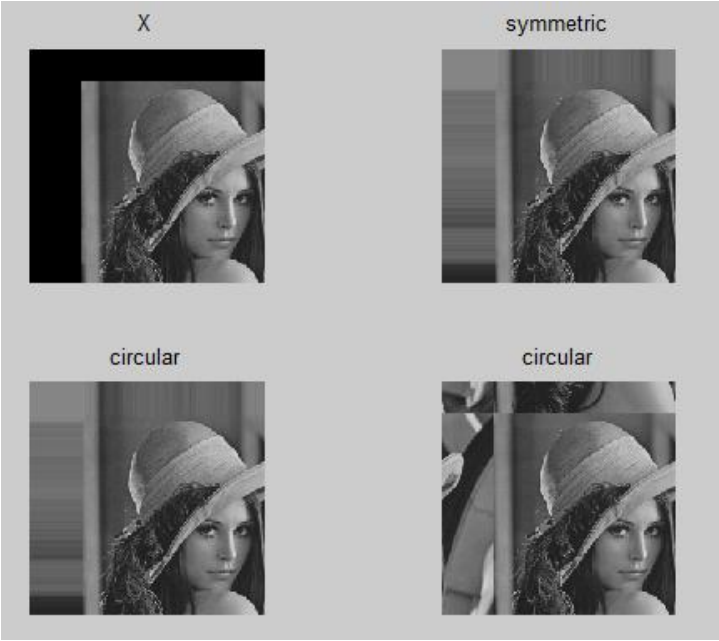

Fig.7 Translating lena image with different boundary conditions

I have tried all types of boundary conditions to see the differences. In case of *symmetric* and *replicate* the space that is empty is filled with pixels having colors of pixels across array border or the nearby pixel colors. In case of *circular* it is considered that the lines and columns are periodic so everything that goes beyond the down border appears on the upper side of the image while everything that goes across the right border appears on the left side of the image.

Exercise 1.3.

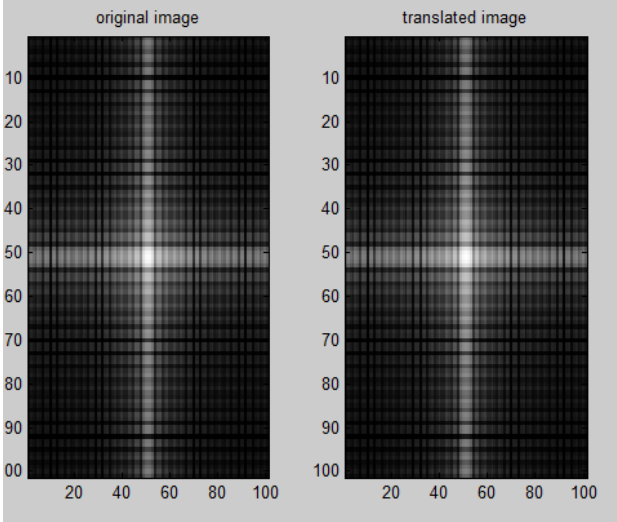

Fig.8 Power spectrum for the image with white square before and after translation

In case of translating the image with white square that was created for the previous exercise, the power spectrum of original image and the translated image seems to be exactly the same.

But, if looking at the table 5.3 from the book at the *shifting theorem*, the Fourier transform of the shifted image is the Fourier transform of the original image that is multiplied by a phase term. So it is the Fourier transform of not translated image with change of phase.

The shift theorem is: $f(x-a, y-b) \rightarrow \exp[-i(k_x a + k_y b)]F(k_x, k_y)$

Exercise 1.4.

That should be possible by applying the Fourier transform for the image, then apply the formula from the right side of shift theorem and then calculate the inverse Fourier transform.

I have written a Matlab code to take an image, apply the fft2 transform and then apply the shifting theorem for translating the image in the frequency domain.

After making the multiplication of image with $\exp[-i(k_x a + k_y b)]$ I have applied the inverse Fourier transform and then displayed the results.

By only doing these, I got some wrong results and then I have found that I need to convert the coordinates to get the correct spatial frequencies so I used the first entry from Table 5.1 in the course book to make the conversion.

```
Ifft = fftn(I);
Ifftr = zeros(size(Ifft));
Ifft = fftshift(Ifft);

a = -10;
b = 10;

for kx = 1 : size(Ifft,1)
    for ky = 1 : size(Ifft,2)
        Ifftr(kx,ky) = (exp(-1i*((kx-1-
size(Ifft,1)/2)*(2*pi/size(Ifft,1))*a+(ky-1-
size(Ifft,2)/2)*b*(2*pi/size(Ifft,1)))))*Ifft(kx,ky);
    end
end

Ifftr = fftshift(Ifftr);
It = abs(ifft2(Ifftr));

subplot(1,2,1);imagesc(I);title('original image');
subplot(1,2,2);imagesc(It);title('translated image');colormap(gray);
```

Fig.9 Code for making image translations using the Fourier method

The code takes the input image *I* and then applies 2D Fourier transform and also shifts the result because I want to have (*0, 0*) coordinates in the middle for easier computation. *a*and *b* are the values to make vertical and horizontal translation. For each pixel with index (*kx,ky*) I substracted the *size/2*to get the correct spatial frequencies indices considering that (*0,0*) is in the middle of image and I have then applied $\exp[-i(k_x a + k_y b)]$ to use the *ShiftTheorem* as it is described in Table 5.3 from the course book.

In order to get the correct spatial coordinates I have also used conversion in Table 5.1 from course book for the spatial frequency entry. That is why I have done the multiplication with *2*pi/size*.

I have at first applied the code to the image containing the white square in the middle. I have at first used values (a = -10, b = 10) for making the translation and it is visible that the square was translated. I have also zoomed to prove that there it is a translation by 10 pixels on the horizontal and vertical, so the algorithm works properly.

I have also tried to see what happens if I translate the image enough to get part of the square out of the image and it looks like it happened the same thing as in case of using option '*circular*' of Matlab function *imfilter*and anything that goes out on one side appears on the other side.

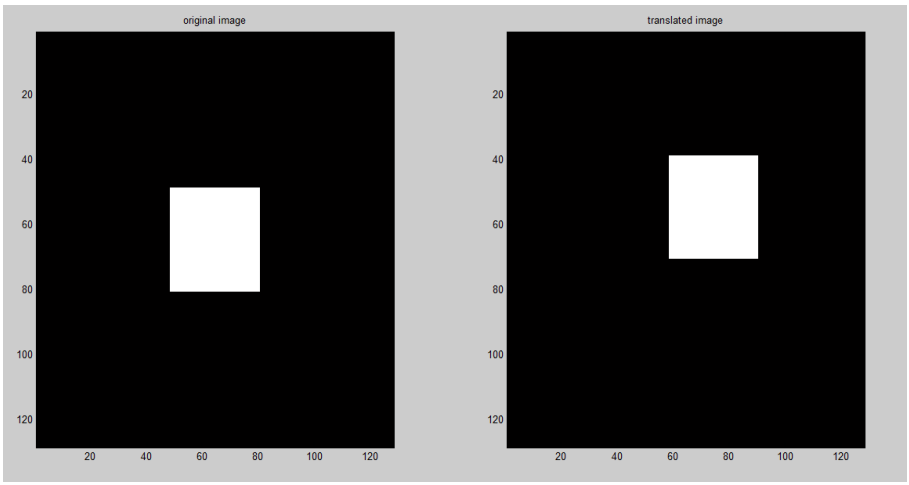

Fig.10 Translation of (a = -10, b = 10) applied using the Fourier method to the square image

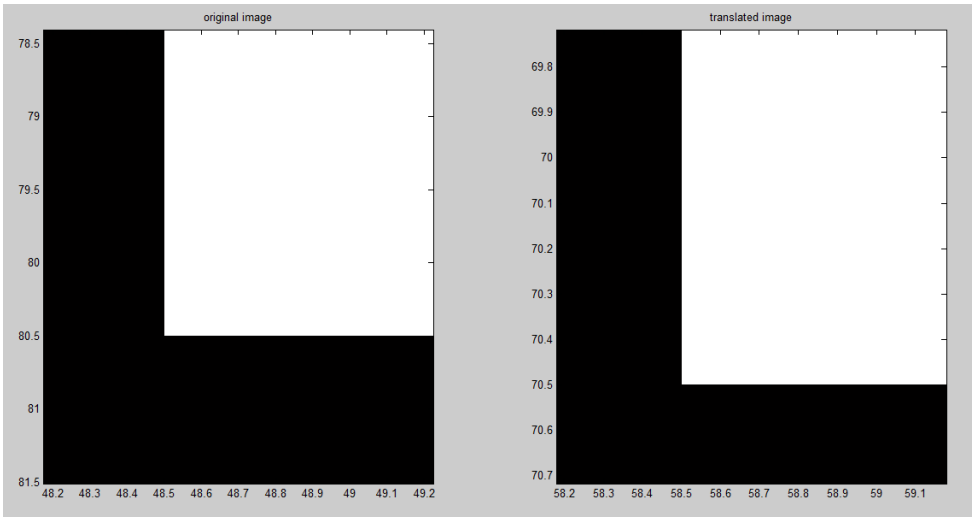

Fig.11 Translation of (a = -10, b = 10) applied using the Fourier method that was zoomed to prove that it is correct

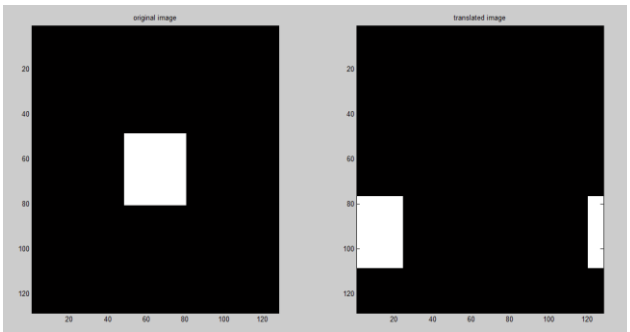

Fig.12 Translation of (a = -100, b = 200) applied using the Fourier method to show what happens when square gets out of the image borders

Fig.13 Translation of (a = -50, b = 50) applied using the Fourier method to the *lena* image

Using the translation for lena image also shows that there is a circular translation of the image. This happens because Fourier transform takes the image as being a periodic signal, where a

period contains the whole image and the next period contains the same image again and so on. The translation in Fourier domain does basically a change in phase of the image.

Exercise 2.1.

In order to make the Procrustes alignment, according to the course book, there are needed the following parameters:

- translation vector $\vec{t}$ which contains 2 parameters: vertical translation and horizontal translation
- parameter *s*which represents how much to scale the image that is obtained through equation 7.20 from course book
- parameter *R* which defines how many angles to rotate the image

So, in total there are needed 4 parameters (2 for translation, and each 1 for scaling and rotation).

Because we need 4 parameters, then *N=2* points are enough for making the Procrusets alignment because each point contains a pair of parameters (x and y coordinates) so 2 points are enough.

Exercise 2.2.

```
I1 = imread('westconcordaerial.png');
I2 = imread('westconcordorthophoto.png');

subplot(2,2,1);imagesc(I1);colormap(gray);
subplot(2,2,2);imagesc(I2);colormap(gray);

cpselect(I1,I2);
```

Fig.14 Code for using the Matlab function *cpselect* for choosing the moving points

In the first stage, I have loaded the images and then used the function *cpselect* of Matlab to select points in the first image that will be considered for making the Procrustes alignment. By doing this, I will also prove that what I have stated at Exercise 2.2 is correct. 2 points in each image will be enough to make the alignment, but they have to be almost same spot in the satellite image to make the alignment correctly.

Fig.15 Chosing the points for calculating the transformation parameters

In the window that was opened, I have chosen 2 points in the first satellite image at 2 crossroads in each image to be easier to figure that they are almost same spots on the maps. In vectors I1 and I2 are stored the coordinates for the 2 pairs of points and I have then read them from the Matlab console and used them in the code to make the Procrustes alignment.

```
movingPoints = [129.7500,92.7500;318.7500 ,81.2500];
fixedPoints = [173.7500, 104.7500; 348.7500, 131.7500];
transf = fitgeotrans(movingPoints,fixedPoints, 'nonreflectivesimilarity' );

%calcualte parameters
u = [0 1];
v = [0 0];
[x, y] = transformPointsForward(transf, u, v);
dx = x(2) - x(1);
dy = y(2) - y(1);
angle = (180/pi) * atan2(dy, dx);
scale = 1 / sqrt(dx^2 + dy^2);

alfa = scale*cos(angle);
beta = scale*cos(angle);
l1 = scale*(dx*cos(angle) + dy*sin(angle));
l2 = scale*(-dx*sin(angle) + dy*cos(angle));

T = [alfa, beta, l1; -beta, alfa, l2; 0, 0, 1];

I1t = imwarp(I1,transf);
subplot(2,2,3);imagesc(I1t);colormap(gray);
```

Fig.16 Code for making the procrustes alignment.

The values for points are stored in vectors *movingPoints* for the points in the first image that will be transformed to make it aligned with the second one, and the vector *fixedPoints* are the corresponding points in the second image and both pairs of points are used to calculate the translation, scaling and rotation parameters.

I have then applied the Matlab function *fitgeotrans*to obtain the transformation matrix for making the Procrustes alignment which is taking the pair of points and calculates the parameters and calculate the matrix as in 7.13 and 7.14 from the course book. To have better understanding what happens I have used Matlab help to see what calculations are done. I have used those equations that I have pasted in the code from the help information to figure out how the parameters are calculated and I have mostly understood it. Then, I have used the *imwarp*function to apply the transformation matrix.

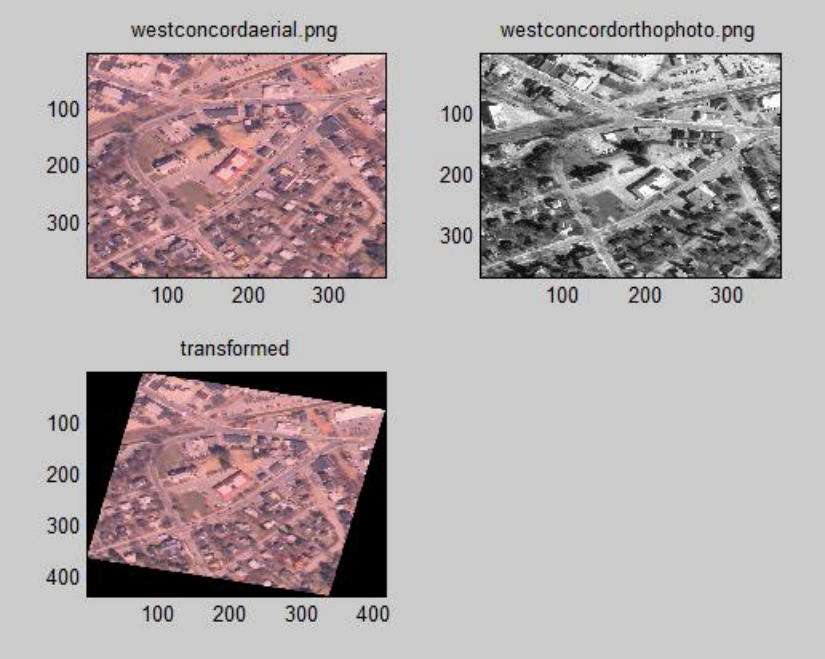

Fig.17 Apply procrustes transformation to align the first png image to the second one

By running the code and displaying the result, the first image was transformed so it is aligned to the second one. It is visible that the 2 large streets and crossroads in upper part of the image have the second orientation and location as in the greyscale image.

It is perfectly visible that I obtained that what Table 7.3 from the course book that Procrustes alignment preserves the same angles and parallelism as in the original image.

Exercise 4.1.

The interest of using homogeneous coordinates is to convert the additions to multiplication which is easier to make. Instead of repeated additions, the whole alignment (Procrustes, affine or projective) turns into simple multiplication with a matrix. This can also lead to faster computation of alignment.

Exercise 4.2.

```
I1 = [0,0;1,0;1,1;0,1;0,0];
I2 = [0.6,0.6; 1.3,0.8; 1,1.6; 0.4,1;0.6,0.6];

plot(I1(:,1),I1(:,2),I2(:,1),I2(:,2));
hold on;
```

Fig.18 Code for Plotting the two quadrilaterals

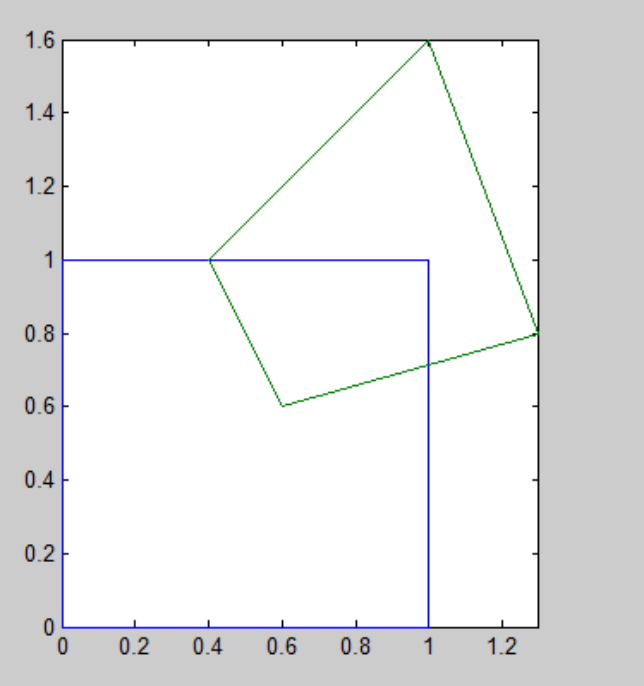

Fig.19 Plotting of the two quadrilaterals before alignment

For plotting the two quadrilaterals I have created two 2D arrays containing pairs of coordinates for the list of points that was given. Then I just simply plotted each dimension in the array and used the function *hold on* to be able to also plot the aligned quadrilateral.

```
transf = fitgeotrans(I1,I2,'similarity');
I1(:,3) = 1;

I1t = I1*(transf.T);

plot(I1t(:,1),I1t(:,2),'r--');

 axis image;
```

Fig.20 Code for applying the procrust alignment.

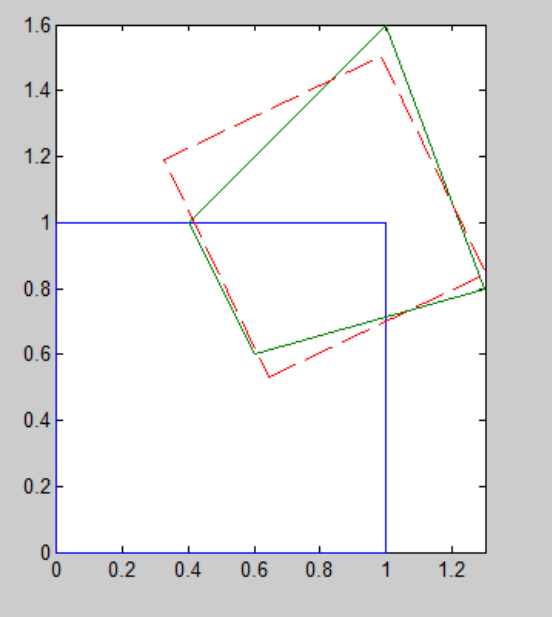

Fig.21 Result for making the procrust alignment

For making the procrustes alignment I have, again used the *fitgeotrans* function of Matlab to get the transformation matrix but this time the *movingPoints*are the points the the first quadrilateral, *I1*, and the *fixedPoints* are points of *I2*. After this, I have done multiplication of the points in the square with the transformation matrix. I have displayed the aligned quadrilateral with red line to be more visible.

After running the code, it is visible that the parallelism and angles are preserved as it is described in table 7.3 from the course book, but the square is rotated and scale.

Exercise 4.3.

For this exercise I used the same code as in Exercise 4.3, just I used option '*affine*' for *fitgeotrans*function to make the affine alignment.

In this case only parallelism of the initial square is preserved, the opposite lines are parallel but the angles are no longer preserved, so the square is turned into a parallelogram.

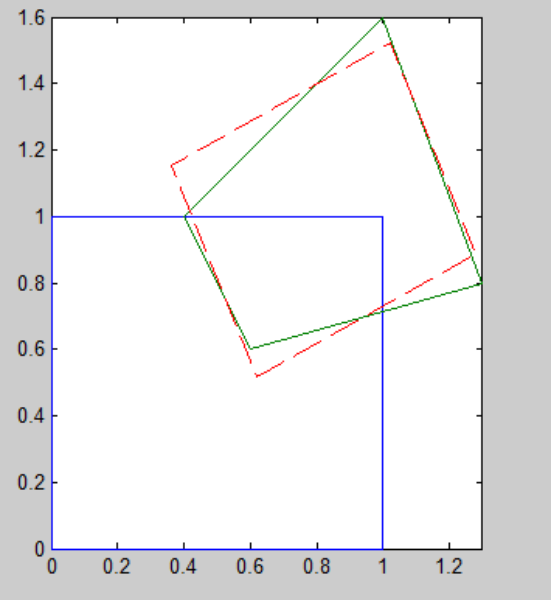

Fig.22 Apply affine transformation to align the square to the green quadrilateral.

Exercise 4.4.

For projective transformation there are 8 degrees of freedom in 2D, so in this case, there are needed 4 points to make the mapping.

```
transf = fitgeotrans(I1,I2,'projective');
I1(:,3) = 1;

I1t = I1*(transf.T);

I1t(:,1) = I1t(:,1)./I1t(:,3);
I1t(:,2) = I1t(:,2)./I1t(:,3);

 plot(I1t(:,1),I1t(:,2),'r--');

 axis image;
```

Fig.23 Code for making the projective alignment

For making the projective alignment I did almost the same as in the previous 2 exercises, but this time I used parameter '*projective*' for function *fitgeotrans*.

For projective alignment, it is obtained a 3x3 matrix for transformation so the points to be aligned (*I1*) needs to be transformed into a 3x3 matrix so a line of 1 is added, just as in the

equation 7.25 from the course book. Because of that the last line in the transformation matrix does not contain , the result for multiplication will not contain values of 1 on the last line.

That is why it is needed a normalization of the resulting matrix and the first 2 lines of the resulting matrix are divided to the last line.

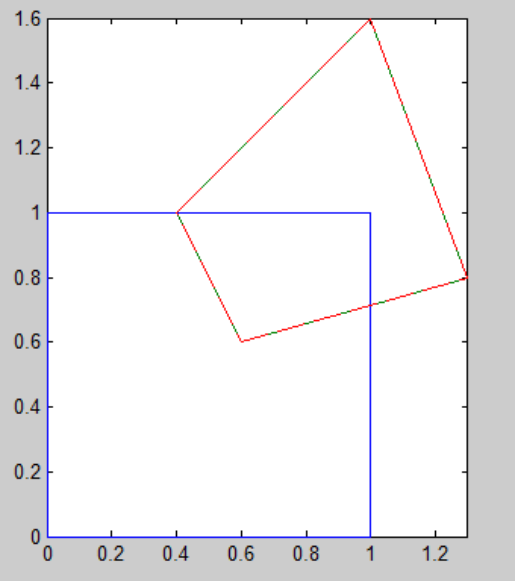

Fig.24 Apply projective transformation to align the square to the green quadrilateral.

By running the code, the aligned quadrilateral looks exactly the same as the green one because the projective alignment preserves only straight lines of the original image.