Image Interpolation, Rotation and Straightening

Report for Lab Exercise 1 for the Computer Vision Course

Anouk Visser 6277209
Rémi de Zoeten 6308694
University of Amsterdam
The Netherlands

November 13, 2011

Contents

1 Introduction

In this report, we demonstrate our findings of two weeks worth of assignments for the course 'Beeldverwerken,' taught by Leo Dorst at the University of Amsterdam.
The various assignments consist of part theory and part Matlab implementation, and focus on the manipulation and interpretation of image data. We were guided by lectures, a teacher provided syllabus and two assignment tutors.

The series of assignments start of by two methods of image interpolation (*nearest neighbour* and *linear*) as interpolation is at the very basis of image comprehension. Next is *rotation*. The basis for rotation is the 2×2 rotation matrix, but this works only for rotation around the origin. So how do we work around this? Then on to a more general set of transformations: the *affine transformations*. Corners are no longer preserved as images can now be sheared, rotated, scaled and even end up in another location. Luckily we still have our straight lines to hold on to, but not for long. The next part of our assignment has us *re-projecting images*. Thanks to the *ginput* function you can click arbitrary points in the image and have this span stretched-out, re-shaped and re-projected. The last part of the assignment that we have accomplished requires us to *estimate a camera's projection matrix*. Here we will try to map an input of three world coordinates to an image that only has two possible coordinates.

2 Theory

2.1 Interpolation

Formally, interpolation is a method of constructing new data points within the range of a discrete set of known data points.
So an interpolation function f can map any *real* value V_r to an appropriate value from a desecrate set D Therefore $f(V_r) \rightarrow V_d$ Where $V_d \in D$

1. $Nearest(x) = F(\lfloor x + 1/2 \rfloor)$
   This holds because $\lfloor x + 1/2 \rfloor$ rounds the x to the nearest Int and $F(y)$ is known and can parse Integers

2. If we want to fit a line $f_1(x) = ax + b$ through both these points, we should solve the following two equations:

   $F(k) = ak + b$
   $F(k+1) = a(k+1) + b$
   Where $k = \lfloor x \rfloor$

   We could therefore also write these equations as:
   $F(x_1) = ax_1 + b$
   $F(x_2) = ax_2 + b$
   Where $x_1 = x$ and $x_2 = x + 1$

3. The found equations can also be written in matrix form as:

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \end{pmatrix}^{-1} \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} (F_1 - F_2)/(x_1 - x_2) \\ (-x_2 F_1 + x_1 F_2)/(x_1 - x_2) \end{pmatrix} \tag{2}$$

   Given in the notes was:
   $a = F(k+1) - F(k)$.
   To get here we will use our a (i.e. $F_1 - F_2)/(x_1 - x_2$ and resubstitue k back in to the formula:

   $F(k) - F(k+1))/(k - (k-1)) =$
   $F(k) - F(k+1))/(0 - 1) =$
   $-(F(k) - F(k+1))) = F(k+1) - F(k) = a$

   The same goes for b. Given in the notes was:
   $b = (1+k)F(k) - kF(k+1)$

   Again, we will check the correctness of our formula:

   $\frac{-(k+1)F(k)+kF(k+1)}{(k-(k+1))} = \frac{-(k+1)F(k)+kF(k+1)}{-1} = (1+k)F(k) - kF(k+1) = b$

4. We can tell from the lecture notes that:
$f(x,l) \approx f_1(x,l) = (1-\alpha)F(k,l) + aF(k+1,l)$
$f(x,l+1) \approx f_1(x,l+1) = (1-\alpha)F(k,l+1) + aF(k+1,l+1)$

   Given that $f(x,y) \approx f_1(x,y) = (1-\beta)f_1(x,l) + \beta f_1(x,l+1)$, we can fill out the formula, that gives us equation 2.3:

   $f_1(x,y) = (1-\beta)f_1(x,l) + \beta f_1(x,l+1) =$
$(1-\beta)(1-\alpha)F(k,l) + aF(k+1,l) + \beta(1-\alpha)F(k,l+1) + aF(k+1,l+1)$
$=$ eq. 2.3.

2.2 Rotation

1. The matrix **R** that performs a counter-clockwise rotation by ø radians in the origin is given as:

$$R = \begin{pmatrix} cos\o & -sin\o \\ sin\o & cos\o \end{pmatrix}$$

2. We can use this matrix to transform a point $(x,y)^T$ in the original image to a point $(x',y')^T$ in the rotated image (when we only rotate through the origin) if we multiply **R** with the point in the original image. This will give the point in the rotated image:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\o & -sin\o \\ sin\o & cos\o \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

3. When we want to perform a rotation about an arbitrary point **c**, we would like to use the rotation matrix **R**. But, we can only use **R** if we rotate through the origin. In oder to use **R**, we will translate **c** to the origin $(0,0)^T$), then we will rotate this point in the origin and at last we will translate **c** back to its original location.

4. **c** itself will not change if we rotate it in the way that is discussed in 2.3. We will prove this algebraically.

   Assume that point **c** is given as:

$$c = \begin{pmatrix} a \\ b \end{pmatrix}$$

   By subtracting **c**, from itself, we will bring it back to the origin, after this we can safely perform the rotation:

$$cTranslated = \begin{pmatrix} a \\ b \end{pmatrix} - \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$cTranslatedRotated = \begin{pmatrix} cos\o & -sin\o \\ sin\o & cos\o \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

   When we translate textbfc back to its original point (we add original point textbfc back to the newly rotated translated version of textbfc), we will see that **c** has remained the same:

$$cRotated = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

2.3 Affine Transformations

1. When trying to build a matrix, c and f 'fall of' due to the matrix being multiplied with a 2×1 matrix.

2. Here we can use the homogeneous coordinates to obtain the c and the f in the formula.

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{3}$$

3.
$$x' = cos(ø)x - sin(ø)y + c$$
$$y' = sin(ø)x + cos(ø)y + f$$

   These formulas rotate, and then transposition.
   ø is the angle of rotation, c is the transposition in the x direction and f is the transposition in the y direction

4. We have chosen for three well known points: the origin, e_1 and e_2
   As this not can be multiplied with a 2×3 matrix, we made these coordinates homogeneous.

   Origin, homogeneous:

$$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{4}$$

   e_1 And e_2, also made homogenous:

$$\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \tag{5}$$

$$\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \tag{6}$$

5. We need at least 3 points because the corners are not preserved. All corners can be inferred given 3 points of a 4-point shape. The fourt point can be inferred.

2.4 Re-projecting Images

1.
$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11}x + m_{12}y + m_{13} \\ m_{21}x + m_{22}y + m_{23} \\ m_{31}x + m_{32}y + m_{33} \end{pmatrix} = \begin{pmatrix} \lambda x' \\ \lambda y' \\ \lambda \end{pmatrix} \tag{7}$$

2. The equations for point $(x_1, y_1)^T$:

   $x'_1 = (m_{11}x_1 + m_{12}y_1 + m_{13})/(m_{31}x_1 + m_{32}y_1 + m_{33})$
   $y'_1 = (m_{21}x_1 + m_{22}y_1 + m_{23})/(m_{31}x_1 + m_{32}y_1 + m_{33})$

   Since for a projective transformation four points need to be transformed, this will give us 8 equations to solve.

3. This transformation should have at least 8 parameters, because there are 8 equations to be solved. Each equations will give us one parameter, thus at least 8 parameters are necessary to perform a projective transformation.

4. The projective transformation matrix only has 8 degrees of freedom. This, because the λ factor can always be divided out. This leaves us with only 8 parameters.

5. We need 12 point correspondences to determine a projective transformation in 2D. This is because we work with homogeneous coordinates (which we did not show in 2). There is also the λ point, which also has it's own equation following from eq. 7. This leaves us with $3 \times 4 = 12$ point correspondences.

6. The vector of unknowns can be given by:

$$\begin{pmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{31} \\ m_{32} \\ m_{33} \end{pmatrix} \tag{8}$$

7. If we set up the equation:

   $x'_1 = (m_{11}x_1 + m_{12}y_1 + m_{13})/(m_{31}x_1 + m_{32}y_1 + m_{33})$

   as

   $\lambda x'_1 = m_{11}x_1 + m_{12}y_1 + m_{13}$
   where $\lambda = m_{31}x_1 + m_{32}y_1 + m_{33}$

We will find:
$0 = -(m_{31}x_1 + m_{32}y_1 + m_{33})x'_1 + (m_{11}x_1 + m_{12}y_1 + m_{13})$
$= -m_{31}x_1x'_1 - m_{32}y_1x'_1 - m_{33}x'_1 + m_{11}x_1 + m_{12}y_1 + m_{13}$

We will do the same for y. That gives us the following equation:
$0 = -m_{31}x_1y'_1 - m_{32}y_1y'_1 - m_{33}y'_1 + m_{21}x_1 + m_{22}y_1 + m_{23}$

If we do this for every point (which we don't because it's a ridiculous amount of work and we will find the same pattern over and over again), we will find we can make a matrix which multiplied with the unknowns matrix, will give us the equations again. This will look like this:

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 & -y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 & -x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 & -y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 & -x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 & -y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 & -x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 & -y'_4 \end{pmatrix} \begin{pmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{31} \\ m_{32} \\ m_{33} \end{pmatrix} \tag{9}$$

8. The trivial solution to the equation $Ax = 0$ is $x = 0$. This is not desirable. Therefore we will try to find the *kernel* of A to find a x that also gives us $Ax = 0$, but now $x \neq 0$.
9. Since we are using homogeneous coordinates with projective transformations, we must take into account that one of the parameters is actually a value that serves as a homogeneous coordinate (which can be divided out). Thus, this particular element of the projective matrix doesn't serve as a parameter.
10. In matlab we can construct matrix A. Through finding the kernel of A we can determine the parameters $m_{11} - m_{33}$. After reshaping the 9×1 matrix to a 3×3 matrix, we have found the matrix of the projective transformation.

2.5 Estimating a Cameras Projection Matrix

1. First of all we will construct two equations for $\lambda_i x_i$ and $\lambda_i y_i$. We will leave out the specific details here, but the found equations are listed below (including the equation for λ_i, which we will use in the next step):

   $\lambda_i x_i = m_{11}X + m_{12}Y + m_{13}Z + m_{14}$
   $\lambda_i y_i = m_{21}X + m_{22}Y + m_{23}Z + m_{24}$
   $\lambda_i = m_{31}X + m_{32}Y + m_{33}Z + m_{34}$

   Now, we would like to have one vector of unknowns (i.e. m_{11} through m_{44}). We will accomplish that by writing these equations as:

   $0 = -m_{31}Xx_i - m_{32}Yx_i - m_{33}Zx_i - m_{34}x_i + m_{11}X + m_{12}Y + m_{13}Z + m_{14}$
   $0 = -m_{31}Xy_i - m_{32}Yy_i - m_{33}Zy_i - m_{34}y_i + m_{21}X + m_{22}Y + m_{23}Z + m_{24}$

For

$$m = \begin{pmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{14} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{24} \\ m_{31} \\ m_{32} \\ m_{33} \\ m_{34} \end{pmatrix} \tag{10}$$

We can now determine the matrix A, in order to get $Ax = 0$.

$$A = \begin{pmatrix} X & Y & Z & 1 & 0 & 0 & 0 & 0 & -Xx_i & -Yx_i & -Zx_i & x_i \\ 0 & 0 & 0 & 0 & X & Y & Z & 1 & -Xy_i & -Yy_i & -Zy_i & y_i \end{pmatrix} \tag{11}$$

This is of course not the complete matrix, for every extra set of points (additional to the already defined point), you'll have to add the described two lines to the matrix.
We're happy for now, we have found a representation that, by using Linear Algebra, will give us the camera projection.

2. We (sort of described) above what happens if we have more correspondences, but we will illustrate it here. If we have for example to correspondences, we will have:

$$A = \begin{pmatrix} X & Y & Z & 1 & 0 & 0 & 0 & 0 & -Xx_1 & -Yx_1 & -Zx_1 & x_1 \\ 0 & 0 & 0 & 0 & X & Y & Z & 1 & -Xy_1 & -Yy_1 & -Zy_1 & y_1 \\ X & Y & Z & 1 & 0 & 0 & 0 & 0 & -Xx_2 & -Yx_2 & -Zx_2 & x_2 \\ 0 & 0 & 0 & 0 & X & Y & Z & 1 & -Xy_2 & -Yy_2 & -Zy_2 & y_2 \end{pmatrix} \tag{12}$$

When more correspondences appear, the matrix will only expand its rows. This means we will still be able to represent all our equations.

3. The exact solution to $Ax = 0$ cannot be found, because there are more equations than there are unknowns.

3 User's guide

In general images are to be loaded correctly and assigned to a variable. With this variable you can consult functions. An example where we load an image:

Listing 1: Loading an image

```
a = imread('cameraman.jpg')
a = im2double(rgb2gray(a))
b = a(:,:,1)
```

Now moving on to rotating this image and prompting it:

Listing 2: Rotating an image

```
imshow(b)
c = rotateImage(b,pi/6,'linear')
imshow(c)
```

To re-project an image, you have to specify specific points. This is done using the *ginput* function. You should specify the points clockwise. Then we re-project the image that is still stored in variable *b* and show the result:

Listing 3: Image re-projection

```
a = imread('flyers.png')
a = im2double(rgb2gray(a))
b = a(:,:,1)
imshow(b)
points = ginput(4)
c = myProjection(b, points(1,1), points(1,2), points(2,1), points(2,2), ...
    points(3,1), points(3,2),points(4,1), points(4,2), 600, 400, 'nearest')
imshow(c)
```

4 Matlab Implementation

4.1 Interpolation

We have written our two interpolation methods in one function called *pixelValue* that takes 4 arguments. The first is an image, the second and third are coordinates and the fourth is the method of interpolation.
The first thing this function does is check wether the given coordinates are in the scope of the image. This is done by the function inImage and we will discus this function later. If it does not fall in the scope of the image, a constant is returned. If is does, with the *switch* statement is selected which method of interpolation is to be used. The procedures of this method are then executed to find the colour that is most appropriate according to the method and the result is stored in the return variable.
The *inImage* function finds the dimensions of the image. Via various statements of comparison, the (boolean) return variable is updated. It returns $True$ if the coordinates are within the contours of the image and otherwise $False$.

We then did an extra exercise with the *profile* function, as described in the assignment. Here are some images of the results.

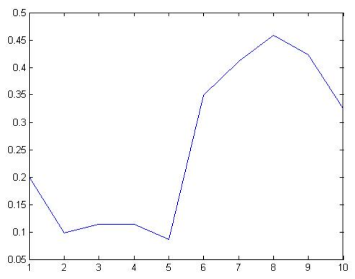

Figure 1: A nearest neighbour interpolation with 10 sample points

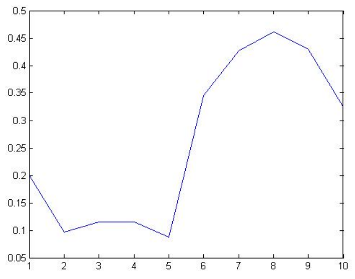

Figure 2: A linear interpolation with 10 sample points

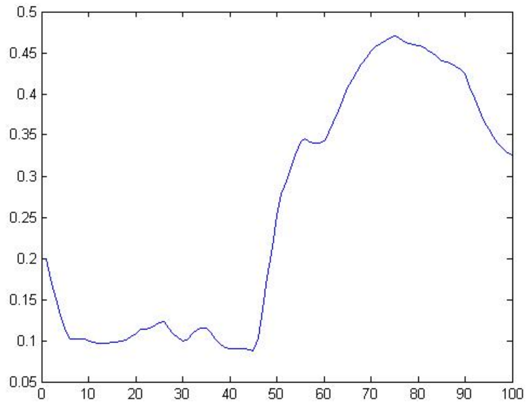

Figure 3: A linear interpolation with 100 sample points

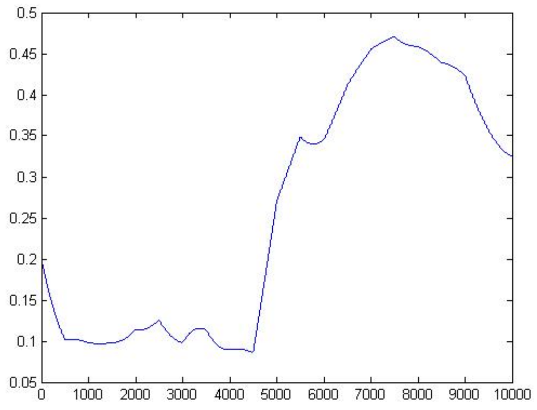

Figure 4: A linear interpolation with 1000 sample points

As you can see, the *nearest neighbour* and the *linear* methods do not show much difference. Also to bee seen is that the sample size maters greatly. The more samples, the more details.
What we also did was take a different approach to the so called 'border problem.' Whenever pixelValue would be called with coordinates that fall out of the picture, in stead of yielding a constant we took the value of the nearest border. Much like the *nearest neighbour* method indeed.
Yet another approach we tried was to look upon the picture as a 'closed space.' You can't simply walk out of a closed space. Nor walk against a border. If you walk long enough in the same direction you are just going to end up in the same place again. Much like here on earth. In this approach, when you 'leave' the image (i.e. coordinates out of the image bound) you are reset to the other side of the image. 10 places out of bounds on the right means 10 places *in* the image from the left.

We have used these alternative problems for image rotation and you can see the results in the next section.

4.2 Rotation

Our function rotateImage takes three arguments: the image that is to be rotated, the angle under which to rotate the image and the method of interpolation.
How this procedure works:
The rotated image is initiated and given the variable name 'rotatedImage' this is a *nul* matrix. To fill the matrix, we will be going through it, point by point, and check what value is to be assigned to the entries of the matrix. Because we are working 'backwards,' from the rotated image to the original image, we have to do the inverse rotation.
So first up we will calculate the inverse rotation matrix. Then will we create a vector of the coordinates of the centre of the image in order to re-adjust every point to the origin for the rotation to be performed. We now use a double *for* loop to iterate over the rotatedImage points, on every iteration executing the following procedures: create vector of the current image points, adjust the centre of the image, do the (inverse) rotation, re-adjust the centre and finally load the colour value of the specific pixel by consulting *pixelValue* with the vector of rotated coordinates.

Finally we evaluated the difference in performance between *linear* and *nearest neighbour* interpolation. For this, we wrote the *tictoc.m* script which yielded the following result:

Listing 4: TicToc

```
linear =

        2.5609

nearest =

        2.1607

ans =

        1.1852
```

Meaning that our *nearest neighbour* method was ≈ 1.18 times faster.
This result, of course, would be more extreme if the rotation function would have been more efficient. This could have been done by, for example, eliminating the double *for* loop. The which is known to be slow in Matlab.

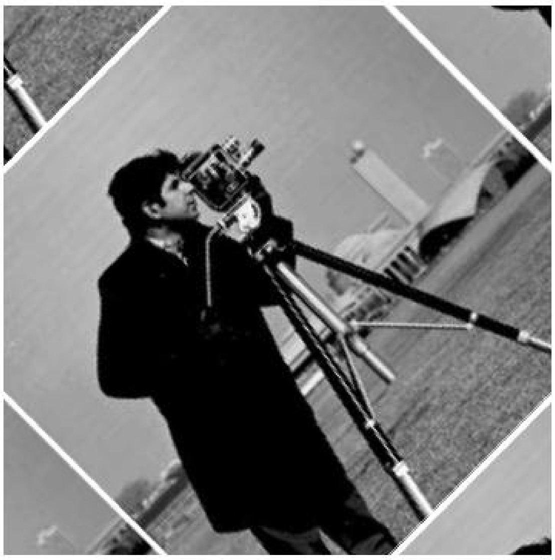

Figure 5: Rotated image of a cameraman with special borders

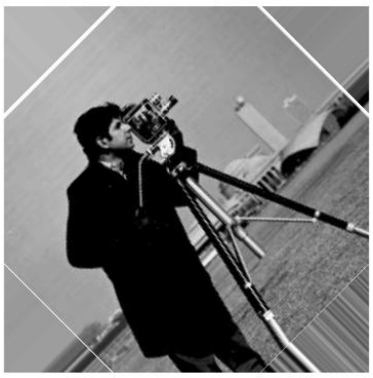

Figure 6: Rotated image of a cameraman with special borders

4.3 Affine Transformations

The function *myAffine* takes ten arguments. The image, 3 sets of coordinates, the number of rows and columns for the output image and the method of interpolation to be used.
In order to perform an affine transformation, we must first calculate the affine matrix. This is done by first initialising the matrix A, that contains the input points, followed by a matrix B, that contains the output points. Since $B = XA$, where X is the affine matrix, we can compute

X by B/A. Now that we have found the affine matrix, we can work in the same way as we did when we were rotating an image.
We will iterate through every point in the new image and for every point we will find the corresponding pixel value from the old image. Because we are working backwards, we will apply the inverse affine matrix to compute the old coordinates that serve as an input to the function *pixelValue*.
Note that we have begun using homogeneous coordinates in every matrix. Otherwise it is impossible to perform an affine transformation.

We can also use the function *myAffine* to rotate images (since that is also a transformation where straight lines are preserved). To rotate the image (cameraman.jpg) 45 degrees counterclockwise we will call *myAffine* as follows:

```
myAffine(cameraman, 178, -74, -74, 178, 431, 178, 357, 357, 'nearest')
```

'cameraman' is the variable in which the image is loaded, the specifics about this assignment can be found in the user's manual section of this report.

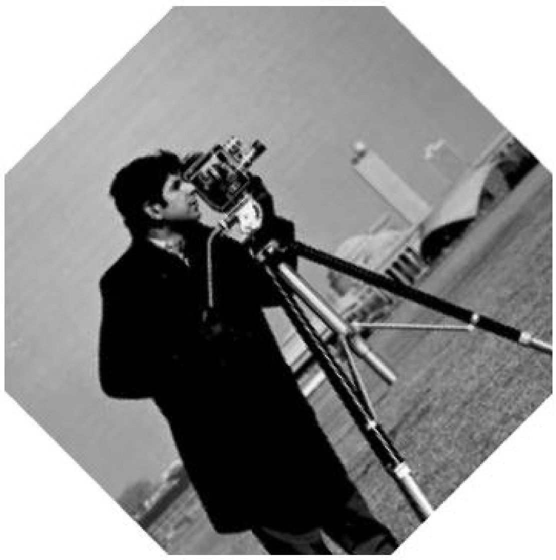

Figure 7: Rotated image of a cameraman through myAffine

4.4 Re-projecting Images

Projective transformations are a bit more difficult than affine transformation. Constructing the projective matrix, in comparison to creating an affine matrix, is much more difficult. Therefore, we have an extra function for that, namely *createProjectionMatrix*. This function takes two arguments: *xy* and *uv* containing the input points and output points respectively.

In the theory section, we have already discussed what a projective matrix should look like. In the code we approach the construction of the matrix in a slightly different way. We create 6 vectors. Two vectors containing the old input and output points, two vectors containing the new input and output points, a vector of ones and a vector of zeros.
Next, we will take these vectors and and put them in two variables, which we will then use to construct the matrix. What you can see is that in the theory section we have a matrix that switches vertically through the 'equation' for a point in the x-direction and then the y-direction. But, the code first gives us all the equations for the x-direction, followed by all the equations for the y-direction. This doesn't make any difference for the results.
It all sounds very promising but we're not there yet. Recall from the theory section that this matrix (which actually is not a projective matrix yet) multiplied by the vector of unknowns gives us a null-vector. In order to find the projective matrix, we must solve this equation. This is easily done. If we take the kernel of the recently found matrix A, we have found the elements of the projective matrix. These elements must be shaped to a 3×3 matrix. This is done in the *myProjection* function.

The *myProjection* function performs the actual projective transformation. Most of the input arguments can be obtained by the *ginput(x)* function, which is a standard built-in function that lets you click on points in the image to obtain x-points from your image. We have used this function to get the second to the ninth argument. The first argument is again the reference to an image and the last three arguments specify the dimensions of the output image and the interpolation method that is to be used.
The function initiates *xy* and *uv* and uses these to call the function *createProjectionMatrix*, which returns a matrix that we reshape manually to the correct format. We then already calculate the inverse of the projective matrix in an attempt to increase the efficiency of the code.
Having done this, we follow the same route as we have done before. We start at the first position in the new image, apply the inverse matrix to the new point to be able to access the pixel value at the old point and assign it to the right point in the image. We do this for every point in the new image by using a double *for-loop*.
This is not yet a complete description of the function, because there is still an unresolved 'problem' that is caused by the homogeneous coordinate. The homogeneous coordinate used to be equal to one in the previous exercise, but now it suddenly got a value. To calculate x, we can't just use our supposed coordinate x, but we'll have to divide it by the homogeneous coordinate λ, to obtain the actual x. Problem solved.

We are very pleased with the result:

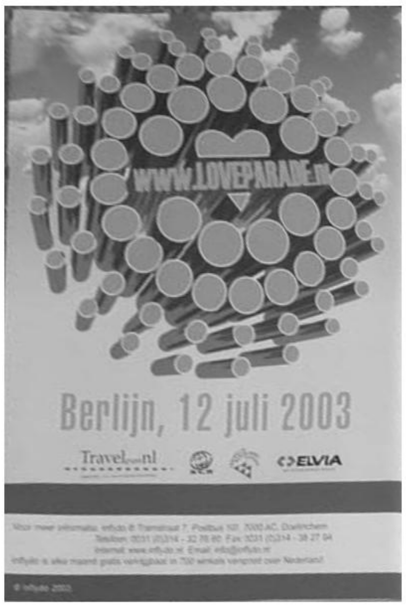

Figure 8: A straightened flyer.

4.5 Estimating a Cameras Projection Matrix

Creating the matrix A is done in a similar way to constructing the A matrix for a projective transformation. The only difference is that here we use an extra input point and we also have three more unknowns. This can also be seen in the theory section.
In contrast to what happened when calculating the projective matrix, an exact solution for M can't be found here. Therefore we have written the function *estimateProjectionMatrix*, that estimates M by using the *svd* function. After reshaping the matrix of now known unknowns, we have found a good approximation of matrix M.

5 Reflection on limitations of the solution, problems and conclusion

There are a few limitation on our solution. First of all, the code is very inefficient. In matlab, *for-loops* are very slow, thus we would have wanted to eliminate them. Unfortunately we didn't have enough time to do so.
Second, we found out much further in the assignment, that we had done some strange x and y values switching at the very start of the assignment. This caused us a lot of confusion further on in the assignment, but it can be even worse when we have to use (pieces) of this code later on in this course.

Overall we have written (well-working) code that can perform basic transformations on an image.