Department of Electrical & Electronic Engineering
Imperial College of Science Technology & Medicine

ISE 1 and EE2 Computing Course

Matlab Assignment: Image Warping

Deadline:
Friday 15th Feb 2002, 12.00 noon.

Please submit to Level 6 Teaching Office:

- A clearly commented listing of your programme (on paper)
- Paper evidence that your programme works (i.e. hardcopy of the results)
- A short account of what you have done
- A floppy disk containing your programme in a form that it can tried immediately

Note:

Do not copy other people's programme. Any violation of this rule will be treated as cheating.

Introduction

In this lab you will write some simple programs using MATLAB.

You will write MATLAB functions to perform the following operations on a grey scale image:

- Rotation
- Shearing
- Edge Detection
- Blurring

Getting Started

Loading the test image

To make things nice and easy for you we've given you an image and a Matlab function to display the image. The image shown in Figure 1 is a 200x320 grey scale image called "clown" for you to work with.

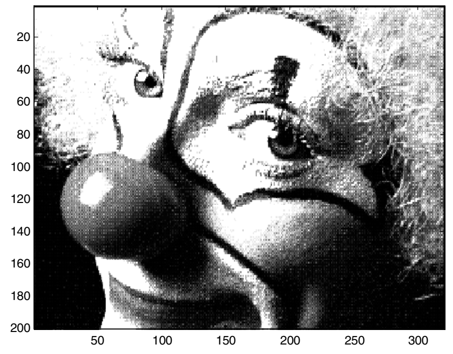


Figure 1 - Grey Scale 200x320 Image "clown"

To load the image into your workspace type:

```
» load clown
```

The Image Format

The image is stored as a 2 dimensional array of grey scale values in the range 0 to 1. To return the grey scale value of the image at co-ordinate (**x,y**) type:

```
» clown (y,x)
```

So typing **clown (20,319)** Matlab responds with:

```
ans =

   0.1554
```

Which is the grey scale value of the image at (319,20).

Displaying Images

The function **show(*Image Name*)** has been given so you can display the images. To display the clown image you previously loaded type:

» show (clown)

Exercise 1 - Image Rotation

Requirements

You should write a function which rotates a grey scale image by *theta* radians, as shown in Figure 2.

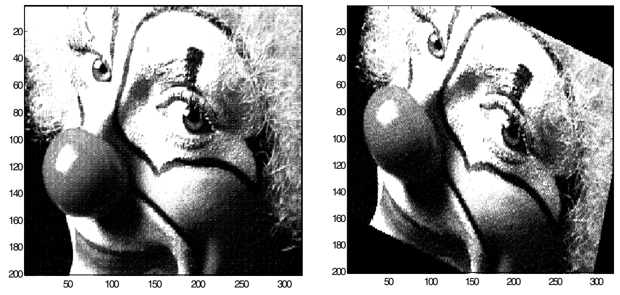


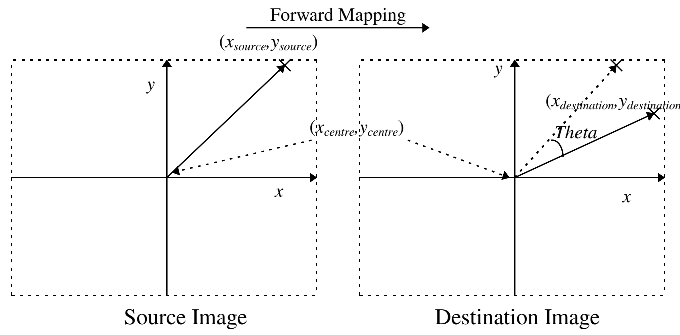
Figure 2 - Original "clown" Image & the image rotated by 0.3 radians

- The function should have the following format:

  function [Out] = rotate(In, Theta)
- The resulting image should be the same size as the original. (i.e. the matrix storing the image should have the same dimensions, so some clipping of the image may occur)
- If a source pixel lies outside the image you should paint it black.
- Use "nearest pixel" only: for example if the source pixel required is (34.43,46.667) you should use the pixel at the location (34,47) in the source image.
- The rotation should be performed about the centre of the image.

Everything You Need To Know About Rotating Images

Just in case your maths is a bit rusty, here's the basics of image rotation:



Forward Mapping
(x_{source}, y_{source})
(x_{centre}, y_{centre})
$(x_{destination}, y_{destination})$
Theta
y
x
Source Image
Destination Image

Equation 1 - Equation for rotation of theta about image centre

$$\begin{pmatrix} x_{destination} \\ y_{destination} \end{pmatrix} = \begin{pmatrix} \cos(theta) & \sin(theta) \\ -\sin(theta) & \cos(theta) \end{pmatrix} \left(\begin{pmatrix} x_{source} \\ y_{source} \end{pmatrix} - \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix} \right) + \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix}$$

The way to use the forward mapping would be as follows:

```
For each pixel in the source image
{
    Work out the destination pixel location using the forward mapping equation.
    Paint that destination pixel with the source image value.
}
```

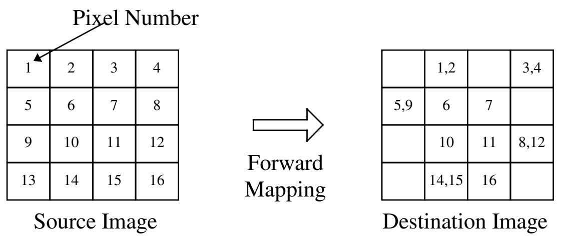


Figure 3 - Using Forward Mapping

But!...

The problem with using the forward mapping directly is demonstrated by Figure 3; Firstly there are pixels in the destination image with more than one source pixel. More of a problem is the fact that some pixels are never written to, leaving the destination image with holes!

The way around this is to use the *reverse mapping* equation in Equation 2. This works out where each destination pixel *came from* in the source image. This uses the inverse of the transformation matrix, which fortunately is easy to work out using the Matlab **inv()** function.

Equation 2 - Reversing mapping of equation 1

$$\begin{pmatrix} x_{source} \\ y_{source} \end{pmatrix} = \begin{pmatrix} \cos(theta) & \sin(theta) \\ -\sin(theta) & \cos(theta) \end{pmatrix}^{-1} \left(\begin{pmatrix} x_{destination} \\ y_{destination} \end{pmatrix} - \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix} \right) + \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix}$$

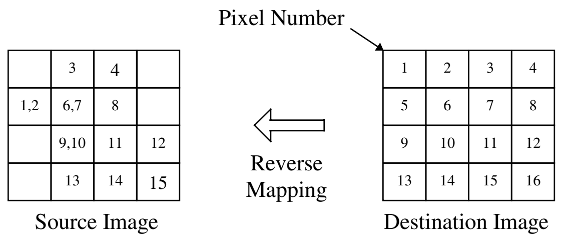


Figure 4 - Using Reverse Mapping

So the way to use the reverse mapping would be as follows:

```
Calculate the inverse transformation matrix
For each pixel in the destination image
{
        Work out where the pixel maps to in the source image, using the reverse
             mapping equation
        Paint the destination pixel with that source pixel value.
}
```

Exercise 2 - Image Shearing

Requirements

You are required to write a function which shears the input image in both the x and y direction and centres the result, as shown in Figure 5.

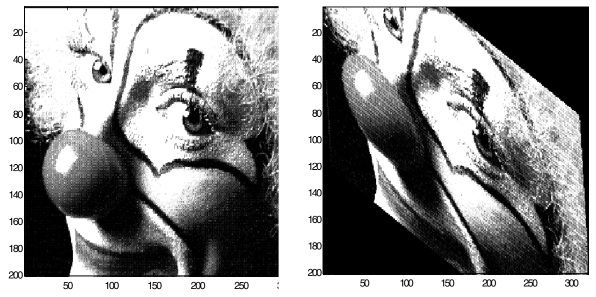


Figure 5 - Original image "clown" and the sheared output using Xshear=0.1, Yshear = 0.5

- The function should have the following format:
  function [Out] = shear(In, Xshear, Yshear)

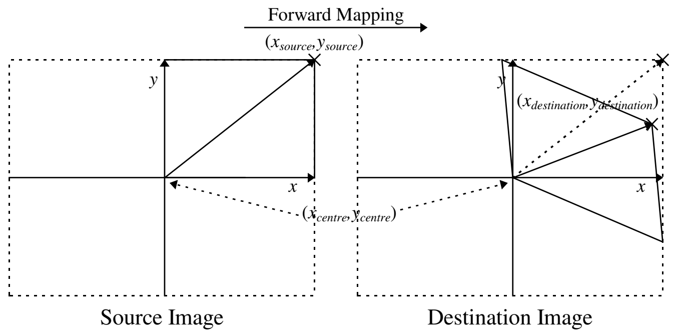
- The resulting image should be the same size as the original. (i.e. the matrix storing the image should have the same dimensions, so some clipping of the image may occur)
- If a source pixel lies outside the image you should paint it black.
- Use "nearest pixel" only, as before.
- You should centre the sheared result (i.e. the centre pixel of the image remains stationary).
- The shear values (Xshear, and Yshear) should be expressed as percentages of the images width and height respectively. Thus when *Xshear*=1, *Yshear*=0

  (0,0) $\Rightarrow$ (-*height*/2,0)

  (0,hieght) $\Rightarrow$ (*height*/2,0)

Everything You Need To Know About Shearing Images

Here's the basics of shearing transformations:


Forward Mapping
(x_{source}, y_{source})
y
x
(x_{centre}, y_{centre})
y
$(x_{destination}, y_{destination})$
x
Source Image
Destination Image

Equation 3 - Equation for shearing the image

$$\begin{pmatrix} x_{destination} \\ y_{destination} \end{pmatrix} = \begin{pmatrix} 1 & Xshear \\ Yshear & 1 \end{pmatrix} \left(\begin{pmatrix} x_{source} \\ y_{source} \end{pmatrix} - \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix} \right) + \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix}$$

As for the image rotation, you will want to use the reverse mapping of the transform to avoid problems with holes in the image.

Exercise 3 - Edge Detection

Exercises 1 & 2 involved image transformation. There was no processing using the pixel values themselves: just simply copying the appropriate source pixel value to the new location in the destination image.

In Exercises 3 & 4 you will process the pixel values in the source image to perform edge detection and blurring of the image.

Requirements

You are required to write a function which detects edges in the image in the horizontal and vertical direction, as shown in Figure 6.

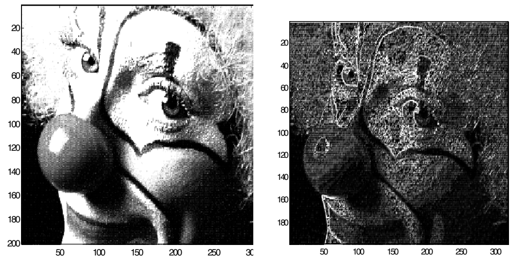


Figure 6 - Original image "clown" and the output from the edge detector

- The function should have the following format:
  function [Out] = dectect(In)
- The resulting image should be 1 pixel less in height and width than the original.
- All the pixel values in the destination image should be between 0 and 1.

Everything You Need To Know About Edge Detection

There are a lot of ways of performing edge detection, but we will opt for the simplest method.

The method will use a *two pass* approach. That means that you will want to look for horizontal edges in the image first, then find the vertical edges and combine the two passes together.

Horizontal formula:

$$Out_{horizontal}(x,y) = abs(In(x,y) - In(x+1,y))$$

Vertical formula:

$$Out_{vertical}(x,y) = abs(In(x,y) - In(x,y+1))$$

abs() gives the absolute value (ignores the sign on a number).

To get the final output we combine these two together:

$$Out(x,y) = \frac{(Out_{vertical}(x,y) + Out_{horizontal}(x,y)}{2}$$

But how does it work?

Consider using the horizontal edge detection formula on the following row of pixels:

Pixel #	1	2	3	4	5	6	7	8	9
Value	0.9	0.9	0.9	0.9	0	0	0.5	0.5	0.5
$Out_{horizontal}$	0	0	0	0.9	0	0.5	0	0	

The horizontal edge detector give us 0 whenever the pixel value is remains constant from one pixel to the next - when there is no edge. When there is an edge (pixels 4→5 and 5→6) the edge detector returns a positive value (always positive because of the abs() function).

This example also helps to explain why the destination image is going to be one pixel smaller in both directions, since we don't know the value for pixel 10 we can't produce an output for pixel number 9.

Why the divide by 2 in the final combining formula? This is to ensure that the output values always remain in the range 0-1.

Exercise 4 - Image Blurring

Requirements

You are required to write a function which 'blurs' the image as shown in Figure 7. This has the effect of removing some of the finer detail, such as on the nose and hair.

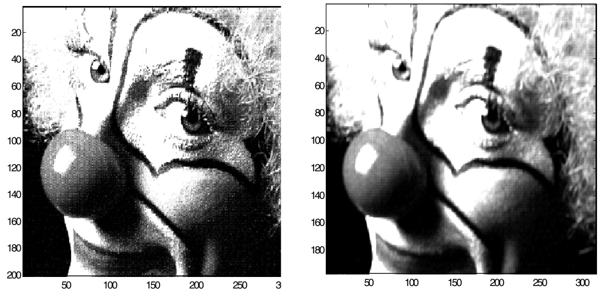


Figure 7 - Original image "clown" and blurred output

- The function should have the following format:
  function [Out] = blur(In)
- The resulting image should be 2 pixels less in height and width than the original.
- All the pixel values in the destination image should be between 0 and 1.
- You should blur the image by averaging each pixel with the 8 surrounding pixels.

Everything You Need To Know About Blurring

Blurring is really a simple type of a more general filtering operation. For the purposes of this lab it is best to think of the blurring operation as an averaging operation.

Blurring formula:

$$Out(x,y) = \frac{\sum_{i=0}^{2}\sum_{j=0}^{2} In(x+i, y+j)}{9}$$

Hint

To get Matlab to work as fast a possible you should always try to use matrix operations.

But how does it work?

Fine detail in the image is due to rapidly changing pixel values from one pixel to the next. By averaging each pixel with it's neighbours this rapidly changing information is smoothed out.