Statistical Image Analysis

Computer Exercise 2: Basic Image Processing

Mats Kvarnström
Department of Mathematical Statistics
Chalmers University of Technology

January 20, 2003

1 Introduction

This computer exercise (CE) consists of two main parts together with a preparatory part. The preparatory part deals with the restoration of the images to be used in the main parts.

In the first main part, we are going to perform some basic linear and non-linear filtering operations just to get a feel for it and in the second part of the exercise we are going to look at thresholding and boundary extraction.

As in CE1, IPT stands for Image Processing Toolbox and everything in `Courier` refers to commands, variables or functions in Matlab.

1.1 Images

The images used throughout this Computer Exercise are the 'Weed Seed' images

- `rum_crisp1.tif` to `rum_crisp5.tif`
- `rum_thyrs1.tif` to `rum_thyrs5.tif`

found on the course homepage under 'Images'. The images are 512 times 512 pixels but have been contracted in the x-direction with a ratio of .68.

Begin with downloading the first image of each kind in your working directory. Load them into Matlab variables using `imread`, either one by one as you work with them, or all at once. You should bear in mind though, that each image takes a considerable amount of memory, so the first method is recommended.

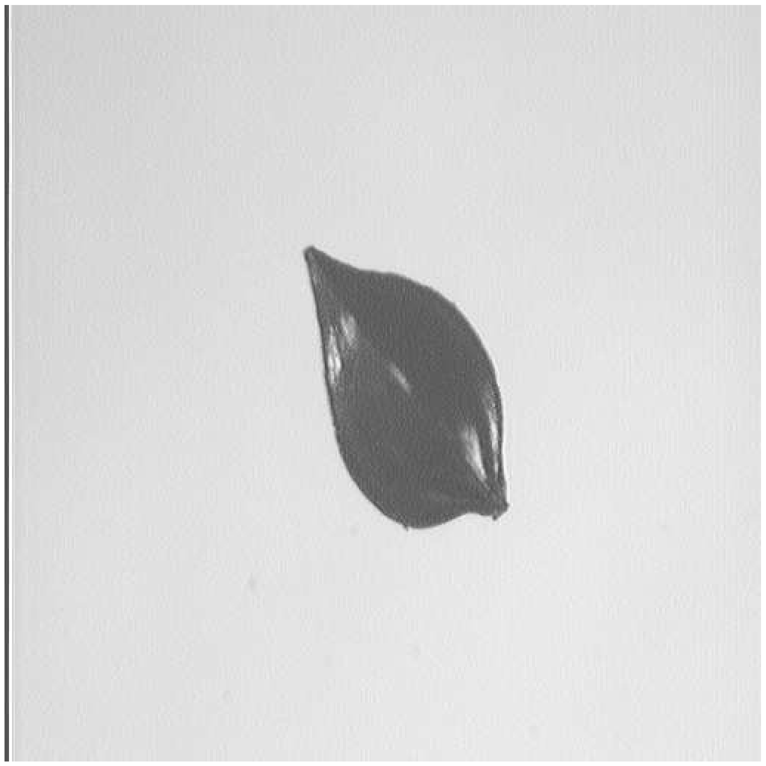

Figure 1: The image `rum_crisp1.tif`. It has been contracted in the x-direction by a factor .68.

2 Part 0: Resampling

Because of the contraction in the x-direction, we need to 'bring back' the image as closely as possible to what it originally looked like. This is called *resampling*. One way to do this is by

1. Define a new image variable `Ir` which is $512/.68 \approx 753$ pixels wide. This can be done by writing `Ir=zeros(512,753);`, setting all pixel values to zero to begin with.

2. For each pixel location in `Ir` assign it a pixel value by linear interpolation from the original image.

After you have done this, keep the central, 512 times 512 part of the image where the seed in located. (It is often nicer to deal with square matrices, and especially if the size is a power of two, 2^n for some n.) What you should do is to implement the algorithm above as a Matlab function, with the contracted image as input and a resampled version as output, in order to be able to resample all of our 'Weed seed' images easily in a later stage.

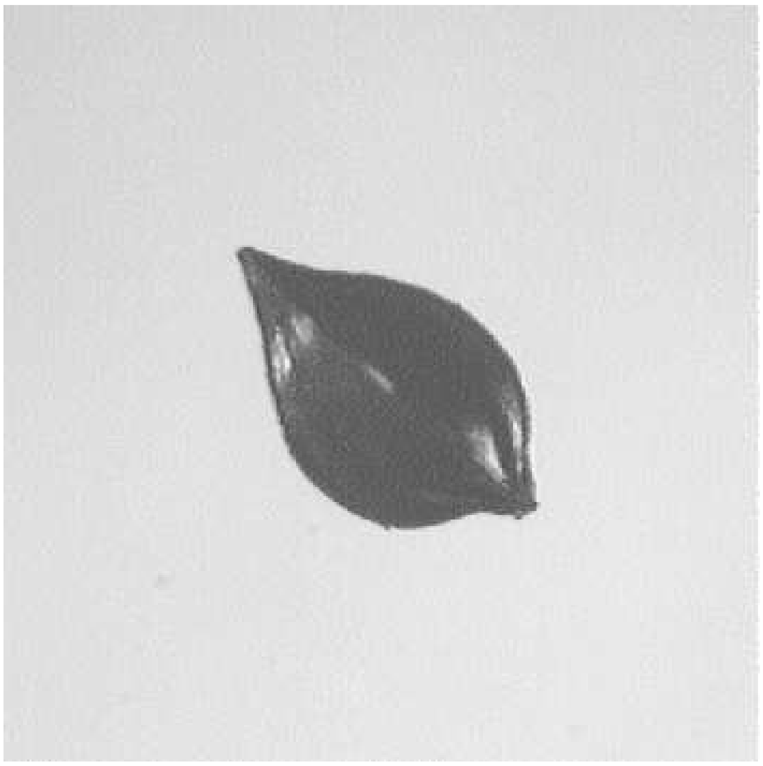

Figure 2: The image `rum_crisp1.tif` after resampling and keeping only the center part.

If you do not know how to write a function or if you want a hint on where to begin, have a look in the Appendix.

The resulting image after resampling and cutting, should look something like the one in Figure 2. Now we have an image that at least have the right proportions.

It is assumed, for simplicity of notation, that the resampled image is called `Ir` from here on.

3 Part I: Image filtering

The purpose of this part is to learn how to perform a filtering operation in Matlab. Furthermore, it is important to get a feel for what the result looks like when you apply a smoothing (lowpass) filter, or an edge emphasizing (highpass) filter to an image.

Filters can also be categorized as linear or non-linear filters. A filter where the output elements are a linear combination of the input elements is called a linear filter, and a filter for which this does not hold is called a non-linear filter.

3.1 Linear filters

The filters in Section 1.2 of the Lecture notes [3] are all linear. To get you started with exploring these, we begin with the 3x3 averaging filter

$$w = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1}$$

In Matlab, start by defining the *filter mask*

```
>>w=[1 1 1;1 1 1;1 1 1]/9
```

To apply this mask to the image `Ir`, use the IPT function `filter2` (do not forget the semi-colon now!) and display it in a new figure:

```
>>Ir_av=filter2(w,Ir);
>>figure(2),imshow(Ir_av)
```

Now you know how to perform a linear filtering with a mask in Matlab. Continue on your own with the 2D Gaussian filter[1] and the two edge emphasizing filters (which also can be considered as derivative approximating filters in the x and y-direction, respectively) found in the Lecture Notes [3] and denote the resulting images `Igauss`, `Ix`, and `Iy`, respectively.

Figure 3 shows the resulting image after applying one of the two edge emphasizing filters. You may stumble into some trouble if you just use `imshow` on `Ix` because it will consist of both positive and negative elements with the main part of the elements centered around zero. To produce Figure 3 the following was used:

```
>>imshow(Ix,[-.15 .15])
```

This tells Matlab to display the values below -0.15, and above 0.15, as black and white respectively, with increasingly darker gray scale values for the elements in between.

3.2 Non-linear filters

An example of a non-linear filter is the non-linear edge detection filter called Prewitt's filter (see for example [2]). It approximates the absolute value of the gradient at each point in the image

$$\sqrt{\left(\frac{\partial I_{i,j}}{\partial x}\right)^2 + \left(\frac{\partial I_{i,j}}{\partial y}\right)^2}$$

by using the derivative approximations from Section 3.1, `Ix` and `Iy`, in the following way:

[1]Take a look at `fspecial` if you have problem producing the 2D gaussian filter mask.

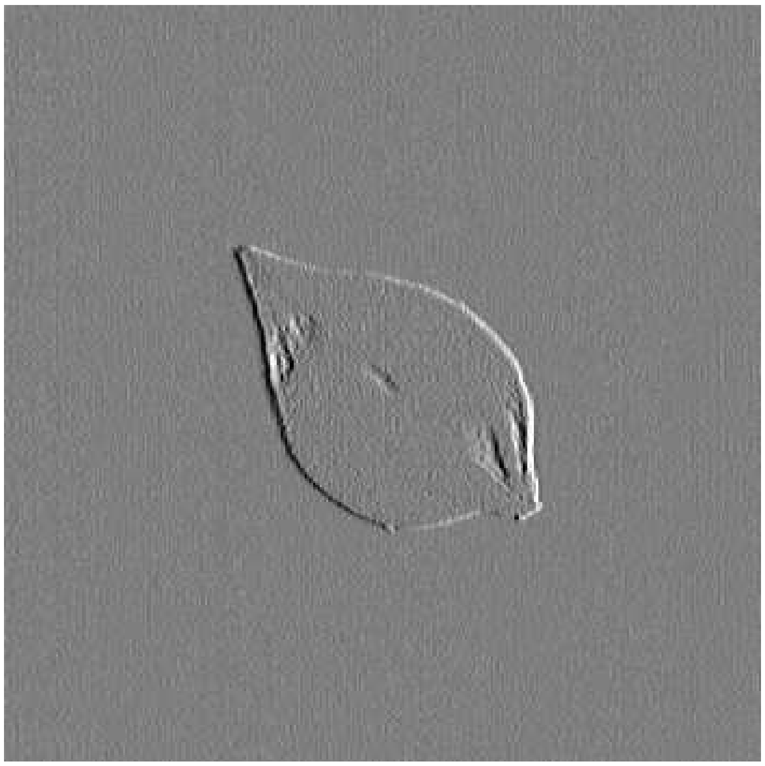

Figure 3: One of the two edge emphasizing filters (guess which one!) applied to the resampled image in Figure 2.

```
>>Ipre=sqrt(Ix.^2 + Iy.^2);
```

As you can see, Prewitt's filter is a non-linear function of the output from two linear filters. Display it (you probably have to use the 'gray scale parameter' in `imshow`), and you will see why it is called an edge detection filter.

3.3 Comments on filtering in Matlab

The Matlab function `filter2` can only be used for linear filters using masks, called FIR-filters[2]. This is by far the most common kind of linear filter in image processing.

The only non-linear filter we have examined yet is Prewitt's edge detector. It was based on the output from two linear filters, so you might ask yourself what to do when this is not possible. The answer is that you probably will have to loop through all the positions in the image and calculate the output element at each position. In the calculation of `Ipre` above, this is actually what you tell Matlab to do when you use the dot in front of the operator `^`, since the dot means that you apply the operation elementwise.

[2]FIR stands for Finite Impulse Response

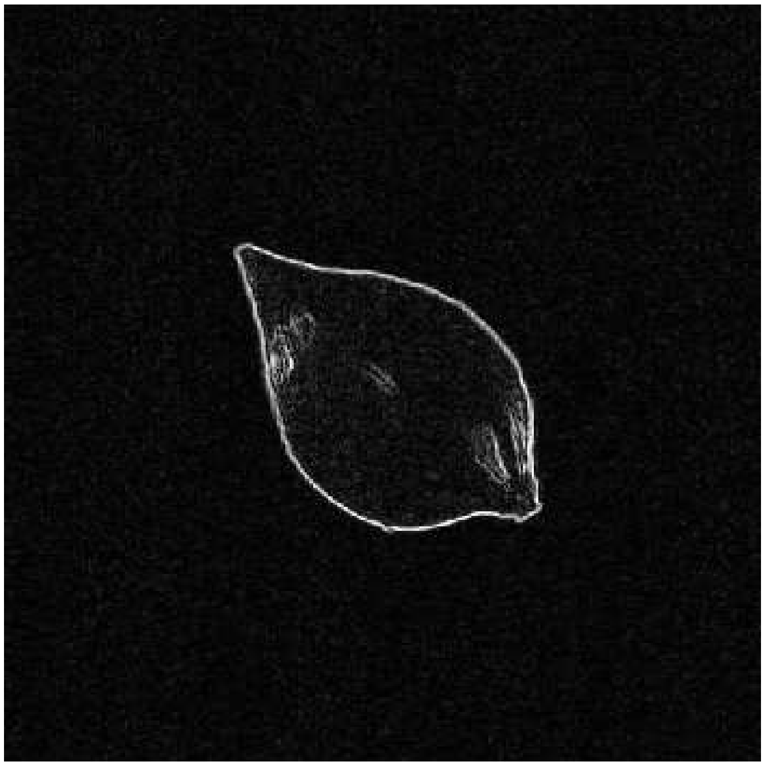

Figure 4: Prewitt's edge detection filter applied to resampled image.

Another issue of importance is what to do along the image boundaries. For example, if you want to apply a linear filter with a mask of size 3x3 to an image of size 512x512, then your mask does not 'fit' along the 1 pixel wide strip along the boundary. By default, `filter2` solves this by artificially 'adding' a frame of zeros around the input image before applying the filter mask, and then keep the central 512x512 part of the image. This is quite bad if the filter is of the edge emphasizing type, because then you introduce an edge along the image boundary (except if the original image had zeroes around the boundary) and the filter will respond accordingly.

So, if it is crucial to have full control of what Matlab does: implement the filter yourself with a loop. It will decrease performance speed, but you will at least have control of what Matlab is doing.

4 Part II: Thresholding and boundary extraction

The goal of this part is to extract the pixel locations corresponding to the boundary of the seed. Furthermore, we would like to know the area, i.e. the number of pixels, the seed occupies in the image.

If we do this for all of our seeds in the different classes, we hope this information can be used as *features* used to discriminate between the classes. See Chapter 2 of [3]. Consequently, our task is to give the ammunition to the subsequent pattern recognition machinery, which we are going to explore in the next computer exercise.

4.1 Proposed solution

First of all, there is hardly ever a best solution to an image analysis problem. The method given here is a bit coarse, by which we mean that it will probably not work unattended, and that it may even have to be modified for some of the images. It is however a decent first step toward a better one and each step may be made more precise later.

1. Thresholding: Choose a suitable threshold value either automatically or manually by inspection of the histogram. After the threshold has been applied we end up with a binary image.
2. Clutter removal: We may have to remove clutter due to a badly chosen threshold. This can be done by some, or a combination, of the morphological operations described in Section 1.4 of [3]. This step should be avoided by choosing a 'good' threshold as possible in step 1, since morphological operations often alters the contour of the boundary.
3. Boundary tracking: Find the largest object in the binary image and extract the pixel locations corresponding to the boundary.
4. Fill out the holes: Some of the images will end up with 'holes' in the seed caused by reflections on the seed while being photographed. If we have extracted the boundary we be able to 'fill out' everything within.

In the next sections we are going to deal with step 1,3 and 4.

4.2 Thresholding

After you have loaded and resampled the image, the idea is to use a suitable threshold to obtain a binary image where pixels with value 1 roughly represent the seed. Look at the figures in Section 1.3 of [3] to see the effect of varying the threshold. Try a number of different threshold values on the different images and answer the following question:

Do you think you could use the same threshold for all ten images with a 'satisfactory' result? Here ,satisfactory means; without too much clutter while at the same time, not destroying the contour of the boundary.

4.2.1 The midpoint method

The answer to the question above is probably a 'No!', so if we do not want to choose a different threshold manually for each image, we need some sort of automatic method to get a good threshold for each new image. Section 3.1 in [1] deals with 3 different methods of finding thresholds for *bimodal* histograms (i.e. a histogram with two main peaks). The simplest method is probably the so called 'midpoint' method and it goes as follows:

Denote the histogram of pixel values (intensities) by $h_1, h_2, \ldots, h_N$, where h_k specifies the number of pixels with gray scale value k (we assume discrete pixel values). Let T denote the threshold.

1. Make an initial guess at T_0, for example the mean, or the median, of the pixel values. The pixels with intensities lower than this value are said to be of the 'lower category'. Those above, are of the 'upper category'.

2. Calculate the mean pixel value of the lower category as

   $$\mu_1(T_n) = \frac{\sum_{k=1}^{T_n} k h_k}{\sum_{k=1}^{T_n} h_k}$$

   and simlilarly for the upper category

   $$\mu_2(T_n) = \frac{\sum_{k=T_n+1}^{N} k h_k}{\sum_{k=T_n+1}^{N} h_k}$$

3. Re-estimate T as the mean of the two categories means

   $$T_{n+1} = \frac{\mu_1(T_n) + \mu_2(T_n)}{2}$$

   and round towards the nearest integer.

4. If $T_{n+1} \neq T_n$, set $n = n + 1$ and goto step 2. Otherwise we are done by letting $T = T_n$.

After the threshold have been calculated write

```
>>Ib=Ir<T;
```

where `Ir` is the resampled image.

The algorithm is fast and easy to implement in Matlab. You are encouraged to do it yourself, but if you do not have the time, though, you may use the version available on the Computer exercise homepage.

Applying this method separately to each of the ten images, results in a very good segmentation and we do not even have to use clutter removal on the resulting binary images.

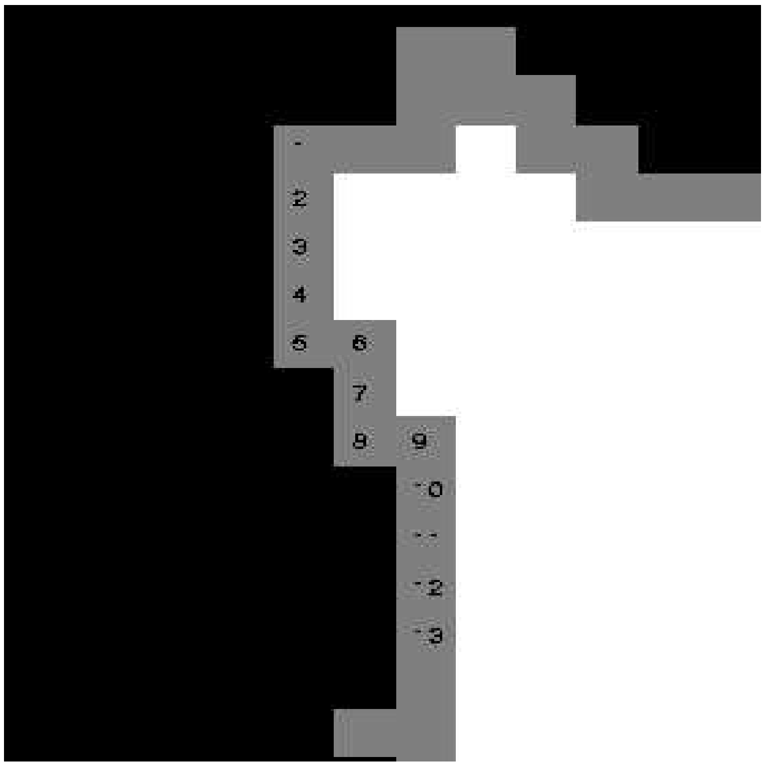

Figure 5: The upper part of one of the seeds. White pixels are the interior and black the background whereas the gray pixels are the boundary, ordered as shown, using 4-connectivity.

4.3 Boundary tracking

In this section we are going to work with 4-connectivity which means that every pixel in the interior of an image is considered to have four neighbours; up, down, to the left, and to the right. Notice that pixels joined by a diagonal are not considered to be connected (i.e. being neighbours) when using this definition.

We are going to track the pixel locations which corresponds to the outer perimeter, i.e. the boundary, of a chosen object using the definition of 4-connectivity. Figure 5 illustrates the principle; the gray pixels are boundary pixels, numbered as shown. What we want, is a vector of connected pixel locations, representing the boundary.

An algorithm doing this, can be found in Section 6.3 of [2] and it is implemented in the Matlab function `contour` found on the Computer exercise homepage. In words, it roughly does the following.

- First we need a boundary point as starting point. This is easy if we do not have any clutter: just use the Matlab function `find` in the following manner

```
>>[y_ones,x_ones]=find(Ib==1);
```

This gives the locations of pixels of value one in the binary image beginning in the upper left and going down each column to the right. Take the first of these as a starting point (x_1, y_1).

- The principle then is to search for pixels in a counterclockwise direction. It does not matter which direction we choose just as long as we act consistently. Now, the second boundary point must lie below or to the right of the first (remember that we only have four neighbours and the way we found our starting point).
- For the rest of the boundary points act like this: Assume that the k:th boundary pixel has location (x_k, y_k). If we came to this pixel by moving to the right (i.e $(x_{k-1}, y_{k-1}) = (x_k - 1, y_k)$), then we start by looking downwards. If we came from above, we start by looking to the left, etc. If this pixel is not an object pixel (i.e. if it is zero), we continue looking in a counterclockwise fashion until we find an object pixel. Let this be our $(k+1)$:th boundary pixel and store its location.
- Continue with this search until we've reached (x_2, y_2) (not (x_1, y_1), can you figure out why?).

4.4 Area filling

It is up to you to figure out how to 'fill out the holes'. Remember that you have the location boundary pixels from the last subsection.

5 Matlab commands used in this exercise

- `conv2`: Performs a two-dimensional convolution. Analog to `filter2` exept that the filter is rotated 180 degrees. Standard Matlab function.
- `filter2`: Performs a two-dimensional linear filtering. Belongs to Image Processing Toolbox.
- `find`: This function returns the indices of the first non-zero element of the input. The input is usually a binary expression.
- `fspecial`: A function used to create predefined filter masks such as the Gaussian kernel function.

References

[1] Mattias Andersson. Weed and crop classification by automated digital image processing. Master's thesis, Chalmers University of Technology, 1998.

[2] C.A. Glasbey and G.W. Horgan. *Image Analysis for the Biological Sciences*. Wiley, Chichester, UK, 1995.

[3] Mats Rudemo. *Image Analysis and Spatial Statistics*. Dept. of Mathematical Statistics, Chalmers University of Technology, 2003.

Appendix: Writing functions in Matlab

If you have not written a function in Matlab, here is briefly how to do it:

1. Start by opening a text editor, for example `emacs`.
2. On the top of the page there should be a so called header, which specifies that the file is a function together with the name of the function and the input and output variables. An example:

   ```
   function Ir=resample(I,rate)
   ```

   Here, `resample` is the name of the function, `I` (the contracted image) and `rate` (the contraction rate[3]) are input variables, and `Ir` is the output variable.
3. Below the header, you should have a few lines of comments. A comment is preceded by a %, and everything on the line to the right of a % is ignored by Matlab. The special thing about a comment right after the header, is that this is what you see if you type `help` followed by the name of your function.
4. Now the actual Matlab code should be written. Notice that all the variables in a function are local. So if you use the same variable name in another function or in the command window as in the function, they will not get effected, and vice versa.
5. Save the function as the function name with the suffix '.m'. The function with the header above should consequently be saved as `resample.m`.

As an example, this is what my function 'resample.m' from Part 0 looks like:

[3]In our task this is always going to be .68, so this input variable is not compulsory

```
function Ir=resample(I,rate)
%Ir=resample(I,rate)
%
%This function resamples the image I
%in the x direction and returns it as Ir (of class double).
%The rate should be .68 for the 'Weed seed' images.


[Ny,Nx]=size(I);
Nx_new=floor(Nx/rate);  %The new size in the x-direction
I=im2double(I);
%Convert to double. This function also automatically brings
%the range down from [0 255] to [0,1]

Ir=zeros(Ny,Nx_new); %Define a new array with zeros

%Initialize
left=1;
%Iterate for each index in the new image variable
for k=1:Nx_new
     %Code...This is for you to do!!
end
```

Notice:

- The header, where the output variable is `Ir`, and the input variables (sometimes called arguments) are `I` and `rate`.
- Everything between the header and the first line of actual Matlab code is displayed in command window if you type `help resample`.
- My frequent(?) use of comments in the code. This is not merely for my care about your understanding of my code, but also for my own sake.

Calling functions

When you use a function in the command window or in another function, you use the syntax as written in the header. The variables do not have to have the same names though, they do not even have to be variables. For example, the following two sequences of Matlab code do the same thing:

```
>>I=imread('rum_crisp1.tif');
>>Ires=resample(I,.68);
```

and

```
>>I1=imread('rum_crisp1.tif');
>>rate=.68;
>>Ir=resample(I1,rate);
```

Scripts in Matlab

A '.m'-file without the function header is called a 'script file' and when executed, it does the set of operations present in the script file, exactly as if you were writing the code in the command window. This is very useful since you do not have to write everything ones again if you change one parameter; you just change the parameter in the script file, save it, and type the name of the script file in the command window.