Chapter 1
Image Processing

A brief Introduction

Image processing is a huge area of application and research; important applications are in medicine, where image processing can increase the quality of an image to improve the clinician's diagnosis. Industrial applications involve object detection, looking for cracks in pipes, or sorting fruit coming down a conveyor belt according to size. Many applications are automated, or at least semi-automated, but some require human intervention to set parameters to achieve near-optimal results. You will experience all of this.

Here we shall use Octave as our toolkit; it is convenient, open source and has a huge library of functions, you will not need to write any code, but Octave is easy to read and understand. One alternative is Open-CV which integrates with Visual Studio; if you are a coder and find you enjoy image processing, then this could be a useful platform with which to develop your skills.

Images are 2D-arrays of 'pixels', where each pixel may correspond to one or more bytes; a color image, e.g., RGB will have 3 bytes per pixel, one per color channel. Octave allows us to process each channel independently, or together based on our needs. A greyscale image (no color) will have one byte per pixel, so its values will range from 0 (black) to 255 (white). Sometimes we shall work with normalized greyscale images where black is 0.0 and white is 1.0. Yep, these are floats.

We shall be covering two areas of image processing; first **image enhancement** which is all about increasing the information transfer from the image to the viewer. Here we shall be using medical images with a view of helping a radiologist make a diagnosis. The second area is **object**

detection which involves finding objects in a cluttered image, and reporting on the objects detected, such as their number and size. You will soon learn that images are corrupted by *noise* which comes from the camera sensor; we this processing.

Figure 1

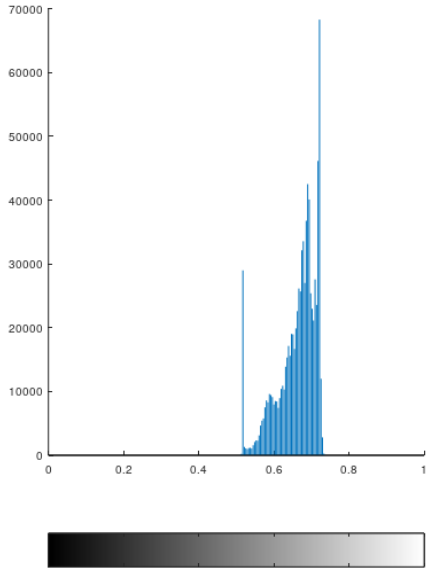


Figure 2

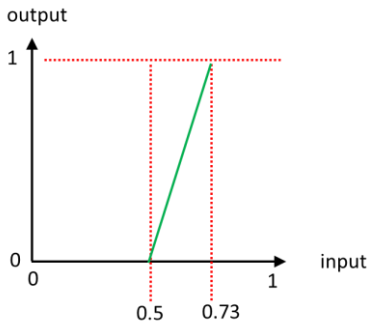
Image Enhancement – Pixel Operations

Contrast Stretching

Consider the thorax x-ray image shown in Fig.1. You can clearly see there is a total lack of contrast in this image making diagnosis impossible. Just looking at the image you can see that there are many grey-ish pixels and almost no black or white pixels.

This is made clear by looking at the *histogram* of the grey-level distribution shown in Fig.2 for this image. Along the bottom of this plot, you find the possible grey level values for this image; it has been *normalized* so these levels range from 0.0 to 1.0. Up the side you find the number of pixels in the image with each of these possible values. So, you can see that the pixels lie in a range about 0.5 – 0.73 which is a small part of the total range 0.0 – 1.0. These pixels have grey-ish values.

It's easy to understand how to enhance this image. The range of pixels needs to be increased from 0.5 = 0.73 to 0.0 – 1.0. So, the value of each pixel has to be individually changed. How this is computed is explained using the graph below.



Input pixel values are mapped to output values by a linear function. Here 0.5 (in) is mapped onto 0.0 (out) and 0.73 (in) is mapped onto 1.0 (out). So the entire range of input values is stretched out

The result of this transformation is shown in Fig.3 with the associated histogram in Fig.4. Looking at the image, you can see a dramatic improvement in its appearance; if images could speak (and they can when you look at them), this one has much more to say. You can see the ribs, collar bones and other features including the heart. Also there is a small cancer nodule, that white circle, in the lung region on the left.

Figure 3 Contrast Stretched Image

The histogram shows that the stretching operation has changed the distribution of grey values which are now spread over the entire available range, 0.0 – 1.0. So the image is being used more optimally to convey information to the viewer. All is not perfect, some pixels are not present in the image as shown by the gaps in the histogram.

Perhaps you could think of a way to improve this, using some sort of interpolation to fill in the missing pixels?

Automatic Contrast Stretching

It is always interesting to see if when a human has made a decision on parameter values, whether this could be automated by some computer algorithm. For contrast stretching this is straightforward. A program could look at all the grey values in an image and find the lowest and the highest. It would then map these values onto the full range of 0.0 – 1.0. Octave has a function

```
stretchlim(image);
```

which returns the lowest value (+1%) and the highest value (-1%). This range can then be used by a second function

```
imadjust(I,[lowIn,highIn],[lowOut,highOut]);
```

to do the actual mapping. This automation clearly corresponds to the discussion above.

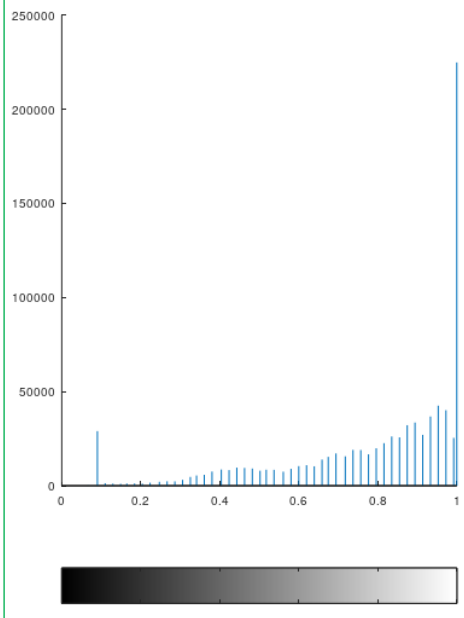


Figure 4 Histogram of Contrast Stretched Image

Histogram Equalization

We have seen how contrast stretching *implicitly* changes the histogram of grey values. This raises the question – can we *explicitly* change the histogram to improve the image. The answer is yes, and this is 'histogram equalization'. The basic idea is that an image communicates *information* to the viewer, and that pixel grey values do this communication. So how can we make this optimal? Well, the first thing is to make sure we use the entire range of grey values, as discussed above. The second thing is to try to change the grey values so the number of pixels with each grey value is about the same. This ensures that each grey value is being used equally as often, and so the maximum amount of information is being communicated. Here's an example applied to the 'Lena' image.

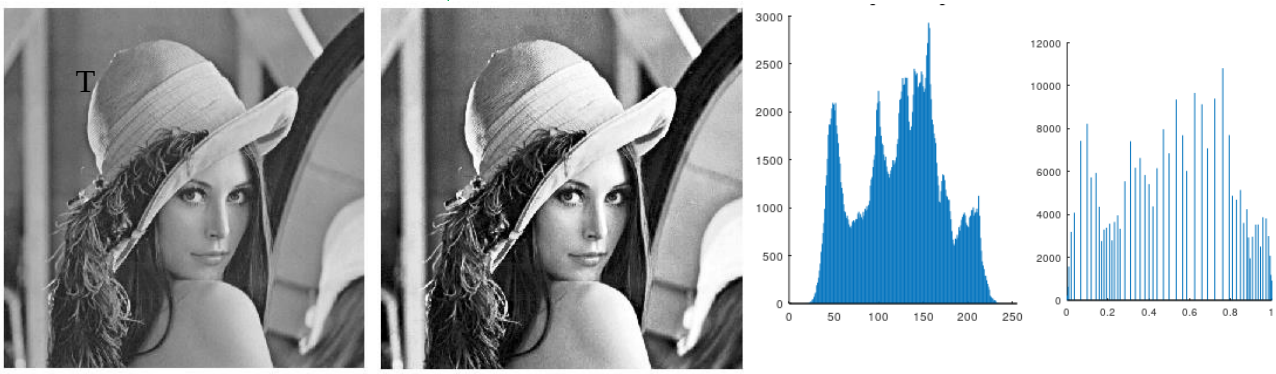


Figure 5 Histogram Equalization: Left image before and after, right histogram before and after

The results are not as spectacular as one would hope; while contrast stretching has been obtained, the numbers of pixels with each grey value is certainly not the same. Comparing the two histograms you may see that the peaks and troughs have been smoothed out, and that the histogram has been *partially* equalized.

Histogram equalization is popular in many areas of image processing, especially industrial applications. One area where it is not widely used is in medical applications since it is a very 'brutal' transformation, often producing large areas

of very bright or very dark pixels. You can see this in Fig.6 where we have applied histogram equalization to the thorax image from Fig.1.

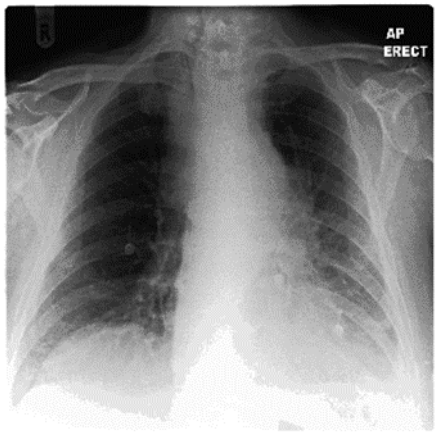


Image Enhancement – Spatial Filtering

The image processing operations we have seen so far have worked on *individual pixels*. Now we turn to a class of operations which process *regions* of pixels. Let's see how the approach works using a diagrammatic representation of an image. The computation works like this; the **kernel** is scanned across all pixels in the image from top-left to bottom right, and at each pixel there is a computation. For the example shown, the *inputs* to the computation are the 9 image pixels lying underneath the kernel cells, and the *output* from the computation is located at the centre kernel cell in the output image.

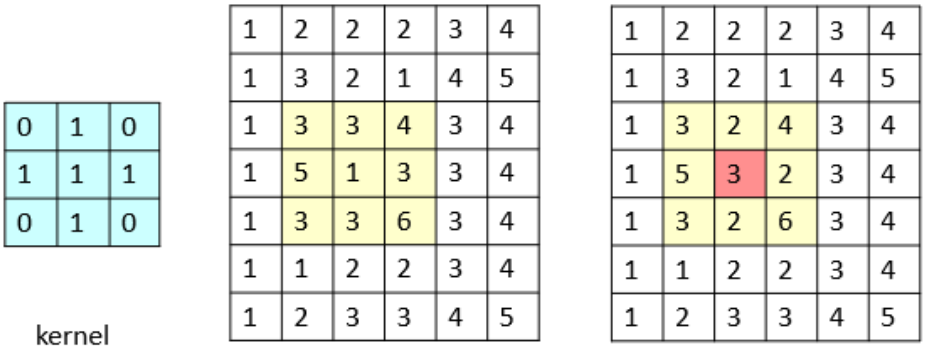


Figure 7 Convolution with a 3x3 kernel

Here the kernel has scanned over part of the image and is sitting on top of the yellow shaded area. The computation is simple; each kernel number is multiplied by the underlying pixel value, and the results are summed. Then we divide by the sum of the kernel values. So, we have

(0x3)+(1x3)+(0x4)+(1x5)+(1x1)+(1x3)+(0x3)+(1x3)+(0x6)

which sums to 15, and we divide by the kernel sum (5) to give us 15/5 = 3 which is the output pixel, shown in read in the output image. This process is called *convolution*.

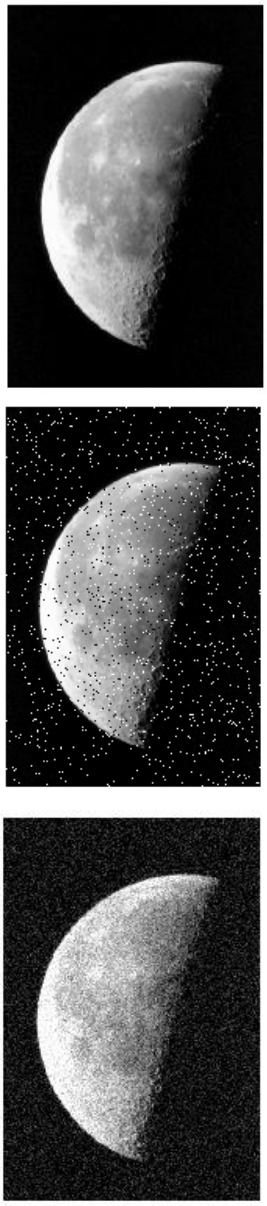


Figure 8 Top, original image, centre with salt and pepper noise, bottom with Gaussian noise

Image Noise

Spatial filtering is mainly used to remove noise in an image. It turns out that noise is the major headache in successful image processing; removal of noise is often the first stage in an image processing pipeline of operations. Where does noise come from? First there is 'salt and pepper' noise which comes from 'dead' pixels in the imaging device, these pixels are permanently black or white due to defects in the camera sensor. Second there is Gaussian noise, a random variation of pixel values around an expected value, this noise follows a normal distribution. Both types of noise will be explored below and are illustrated in Fig.8.

Filtering with the 'Mean' Filter

This is very much like the example above, except that all the kernel values are set to 1.0, and we divide by the sum of these values. In general, the kernel can have size NxN (where N is always odd, so that there is a central 'output' pixel) and the kernel values are all set to

$$\frac{1}{N^2}$$

so the sum of the kernel values is 1.0. E.g., for a kernel size 3x3, its values are set to 1/9.

Applying this filter to the images in Fig.8 gives the results shown in Fig.9 where there is a clear improvement in the image quality, though the salt-and-pepper noise is still visible on the surface of the Moon. There is another filter, the 'median' filter that does a better job at removing salt-and-pepper noise, we'll see that shortly. Also, if you look carefully, you will just be able to see some blurring (smoothing) at the edge of the disk. Here is the kernel used in this example

```
0.11111   0.11111   0.11111
0.11111   0.11111   0.11111
0.11111   0.11111   0.11111
```

To understand the effects of the mean filter, we shall apply it to a 1D synthetic image containing a single edge corrupted by Gaussian noise. We shall then smooth the image with kernels of increasing size. Here's the original image together with a smoothed image of kernel size 3.

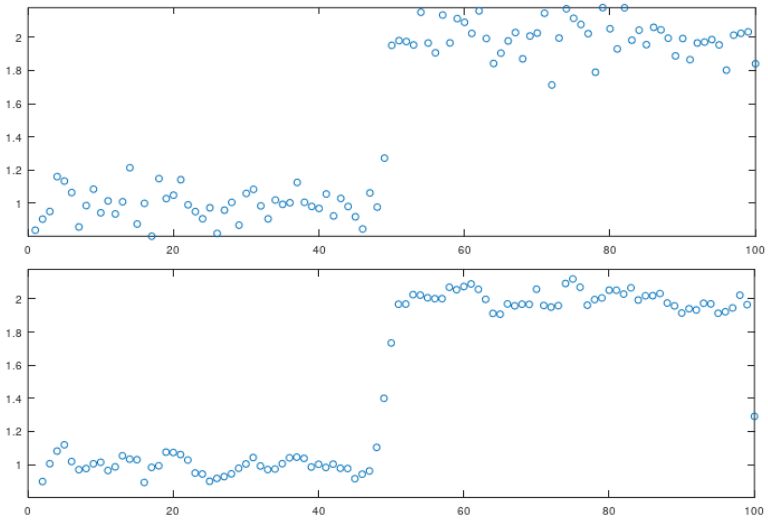


Figure 10 Top, noisy image, bottom results of smoothing with kernel size 3

The original image, before noise was added comprised the left half with a value of 1.0 and the right half 2.0. The effects of noise are clearly seen. After the smoothing, the size of the noise spikes has clearly been lowered, and both left and right halves are smoother. Now let's see the effects of increasing the kernel size to 7.

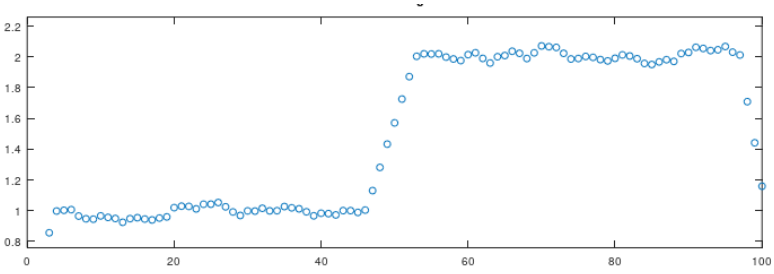


Figure 10 Smoothing the noisy image with kernel size 7

There is much better smoothing here, but something else has changed, the central edge in the image has become less steep

Figure 9Images from Figure 8 smoothed with a mean filter with kernel size 3x3

in other words, blurred. It's easy to understand that as the kernel size gets larger, there is better smoothing out of noise, but edges become smoothed too, which is not really desirable. There is a trade-off here.

The Gaussian Kernel

The mean filter sums image pixels under the kernel footprint with equal weight, which means that the information in the central pixel being processed is diluted with information from pixels further away. The Gaussian kernel does better being processed.

The Gaussian kernel is derived from the continuous function

$$f(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

where the centre of the kernel is at *x=0, y=0*. The parameter σ ('sigma') controls how quickly the kernel 'falls off' in space. Here's a couple of examples, both for kernel sizes 5x5.

0.02	0.03	0.04	0.03	0.02
0.03	0.05	0.06	0.05	0.03
0.04	0.06	0.06	0.06	0.04
0.03	0.05	0.06	0.05	0.03
0.02	0.03	0.04	0.03	0.02

0.03	0.04	0.04	0.04	0.03
0.04	0.04	0.05	0.04	0.04
0.04	0.05	0.05	0.05	0.04
0.04	0.04	0.05	0.04	0.04
0.03	0.04	0.04	0.04	0.03

Left Gaussian kernel sigma = 2, right sigma = 3.

There are several things to note. First the kernels are *symmetric*, they have the same values in equivalent horizontal and vertical locations. Second, the values are larger near the centre, the kernels actually have a bell-like shape. Finally you can see that for a small σ the values drop of quickly as you move out from the centre, the opposite for a larger σ.

The Median Filter

The median of a set of numbers is the midpoint of the set when it is sorted from lowest to highest number. This filter also operates on a NxN neighbourhood, and the sorting operation has to be carried out at each pixel. It turns out that this filter is superior to mean or Gaussian smoothing since it

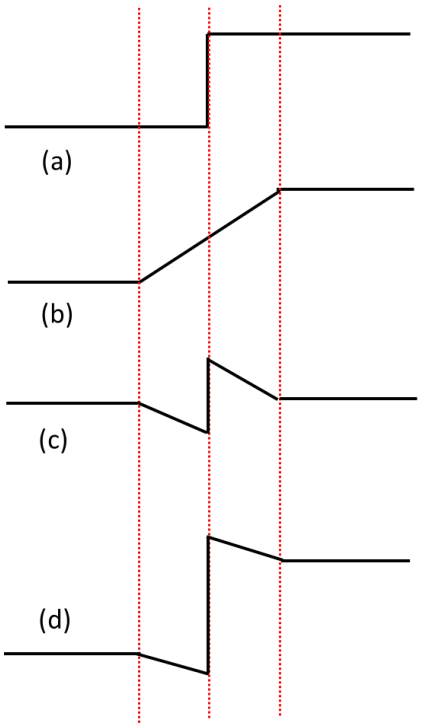


Figure 10 Unsharp Masking: (a) original image (b) smoothed image (c) edge image (d) final image

tends to better preserve edges, and also removes isolated noise spikes such as salt-and-pepper noise. However it is much more computationally expensive than the averaging filters since pixel values in the kernel must be sorted for each centre (output) pixel. Compare this with the averaging filters which involve NxN multiplications and (NxN -1) additions.

Unsharp Masking

This beautiful technique finds its origins in dark-room photography, and was used long before digital image processing. It's aim is to enhance the image by accentuating any edges in the image, producing a more 'crisp' image. It works by smoothing the image with a mean or Gaussian kernel, then subtracting that from the original image to extract edges in the image. This 'edge' image is then added to the original image, with some weighting, to produce the final sharpened image.

This is illustrated with a toy edge (Fig.10): (a) shows the original step edge, (b) after smoothing the extent of which is determined by the kernel size, (c) shows (a)-(b) which has extracted the edge, and (d) is a fraction of (c) added to (a), you can see the edge is accentuated.

Fig.11 shows this applied to 'Lena'. We can also write down these stages mathematically

$$I_{edge} = I_{orig} - I_{smoothed}$$

$$I_{final} = I_{orig} + \alpha I_{edge}$$

where α is typically between 0.0 and 1.0 and after a bit of maths we can simplify this

$$I_{final} = (1 + \alpha) I_{orig} - \alpha I_{smoothed}$$

This requires a smoothing kernel, followed by a subtraction of two image pixels. This can be combined into a single kernel.

Figure 11Unsharp Masking. Images are labelled.

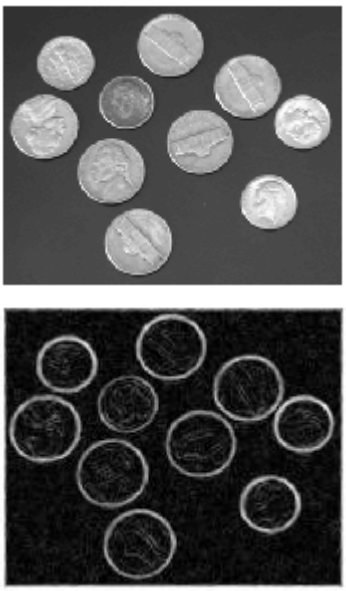


Figure 12 Edge detection applied to coins

Edge Detection

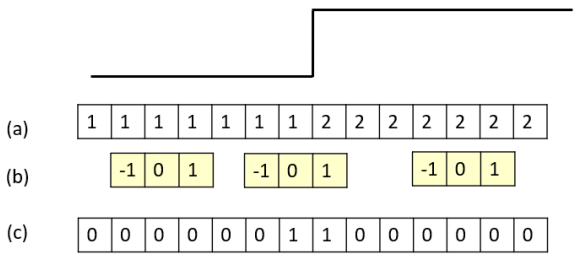
Detecting edges in images is a useful image processing operation and has two main applications; (i) enhancing the image as we have seen, (ii) *segmenting* the image into the objects it contains. We shall see the latter a little later on, for the moment we need to understand how to detect edges. The operation is shown in Fig.12 applied to some coins. The output image grey values are large (white) where there are edges. The algorithm works reasonably well, but it has also detected the edges within the coins. Again, image processing is tricky.

It is understood that the human visual system (HVS) contains a bank of edge detectors tuned to edges and lines of various widths and orientations. The outputs of these are used for object recognition.

Let's design a convolution kernel which detects edges, and let's work in 1D. In regions where the grey level is constant, e.g., a load of 1's (no edge) then the detector should output zero. In other words, the kernel should take the *difference* between nearby pixels. Here's a good candidate

-1	0	1

This will subtract the left pixel value from the right pixel value and output the result at the centre pixel. Let's see this working on a toy image



At the top is a pure edge, the left and right pixel values are shown. In the middle three positions of the kernel are shown

the first outputs 0, the second 1 and the third 0. So where there is no edge, the output is 0, the edge outputs a 1 on both sides of the edge. So if we look at the output image, it would be black with the edge showing as white, rather like in Fig.12.

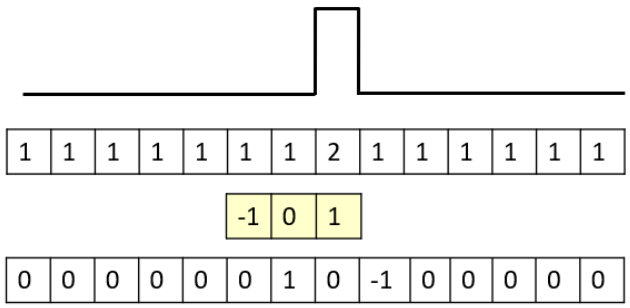
For 2D images we need a difference operator in both directions. One old workhorse is the Sobel operator, which is a pair of 3x3 kernels,

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

The left kernel finds the edges in the horizontal direction (columns) and the right kernel finds the edges in the vertical direction (rows). The Octave function **`imgradient(image)`** calculates both, and combines the results to give the magnitude of the edge value at each pixel in the image. That's what you see in Fig.12.

Edge detection operators are *gradient* operators and respond to changes in image grey level height. Think of roads of various gradients, such operators (in physical space) would tell you the gradient of the road. Unfortunately gradient operators respond rather well to noise, as you can see in the toy problem below which shows a single salt-and-pepper spike on a flat background and the result of our simple edge detector.



We've been here before and so we understand the problem. Before we apply the edge detector kernel, we must first smooth the image to reduce the noise, ideally using a median filter. Of course, too much smoothing will also blur the edges, and may cause some edges to run into each other.

Segmentation

Segmentation is the process of extracting objects or features from an image. Inspection of electronic circuit boards may need to check that all components are present, thus the components (chips, resistors, etc) need to be identified in the image. A simple example is shown in Fig.13 where the original image comprises a number of coins, the objects to be detected in an image. The goal here is to identify all coins in the image, give them a label, and calculate their areas. The image has been successfully segmented into a background layer (cyan) and each extracted coin has been labelled with a color. You can see that one coin has not been segmented exactly.

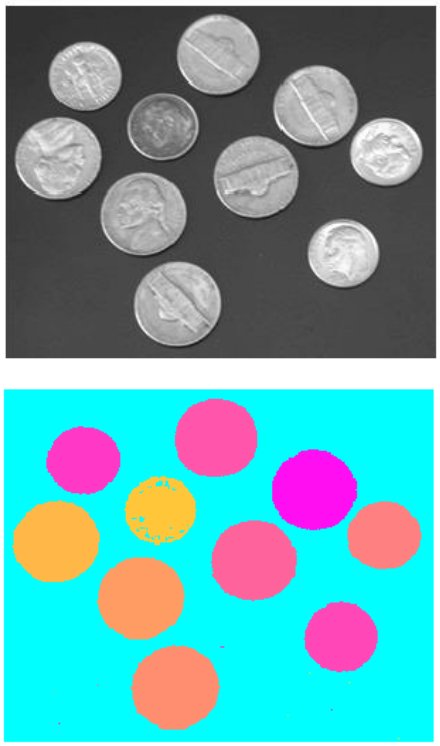


Figure 13 Segmentation: Top original coin image, bottom segmented and labelled coins

Segmentation is one of the trickiest areas in image processing and is still the subject of research, its success often depends on the quality of the input image. For industrial situations this is relatively straightforward where the engineer has some control over environmental variables (such as lighting). For remote (satellite) sensing, e.g., of land usage, this is often difficult, since the image quality is dictated by the sensing device and procedure. There are two broad approaches to segmentation (i) using edges (ii) using regions. We shall look at the latter here.

Interactive Thresholding

Think about a grey level image, Fig.13; this comprises pixels with values in the range 0-255. Now think of a segmented image which we can write as

segmented image = objects AND background

So, a segmented image is a *binary* image where the

background is 0 (black) and the objects are 1 or 255 (white).

How can we create such a binary image? We can draw on our understanding of histograms of grey values. Look at the histogram of the image in Fig.13. What does this tell us? Well we have two peaks, showing there are a load of darker pixels and a load of lighter pixels. Clearly the dark pixels form the background, and the lighter pixels are the objects we are segmenting out. All we need to do is to choose a *threshold* grey value to separate pixels in these peaks out. All pixels whose grey value is less than the threshold are assigned to binary 0 in the output image, those greater are assigned to 1 in the output image. Results for a threshold of 120 are shown in Fig.14. The result is close but not perfect. Interactive thresholding allows the user to tune the thresholding for a particular situation (image type, lighting, etc) so successive images may be correctly segmenting. But of course we would like to automate this process.

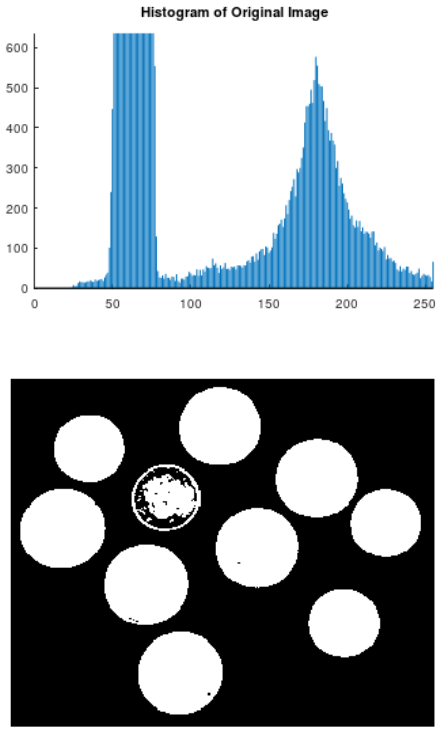


Figure 14 Histogram of image from Fig 12 with results of thresholding at 120

Automatic Processing

Here we outline an algorithm which could find the threshold automatically, and so adjust to the input image.

1. Estimate a good threshold T
2. Segment into two regions A and B
3. Compute the means of each region m_1 and m_2
4. Compute a new threshold
$$T = \frac{1}{2}(m_1 + m_2)$$
5. Repeat steps 2 to 4 until the change in the threshold is less than a specified value

Otsu's Algorithm

This method of computing the optimal threshold partitions the image pixels into two sets. Say there are *L* total grey values in the image. Then pixels are put into one set with values [0,1,2,…,k] and the other set with the remaining larger values [k+1, k+2, …, L-1]. Otsu then calculates a *variance*.

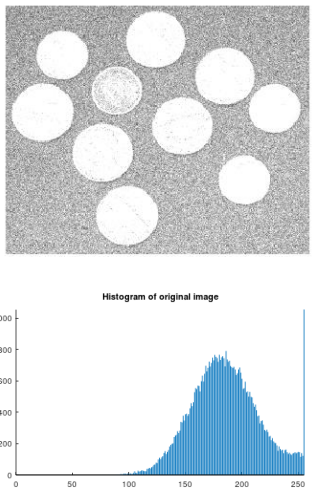


Figure 15 A noisy image and its histogram, Impossible to segment

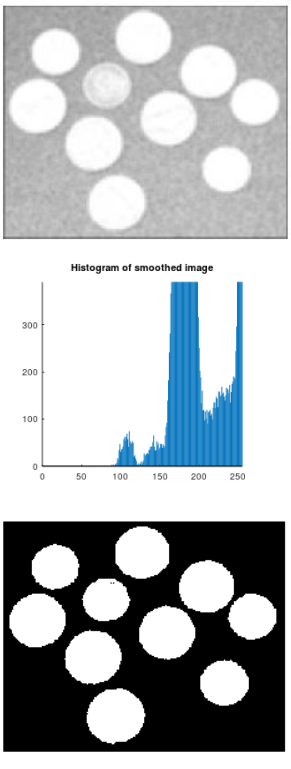


Figure 16 Segmentation by pre-processing smoothing followed by Otsu's algorithm

Let's refresh ourselves what variance means. If you open a can of peas, they all have about the same mean size, differences in size from the mean is small; so we have a low variance. But look at the heights of folk in the lab; there is a well-defined mean (no-one is 3m tall) but there is quite a bit of spread in height, so quite some variance.

Now Otsu looked at the variance of the grey values of the pixels in the two initial sets, but more importantly, he looked at the variance *between* the two sets, and he argued that the best segmentation would *maximize this variance.*

Segmenting Noisy Images

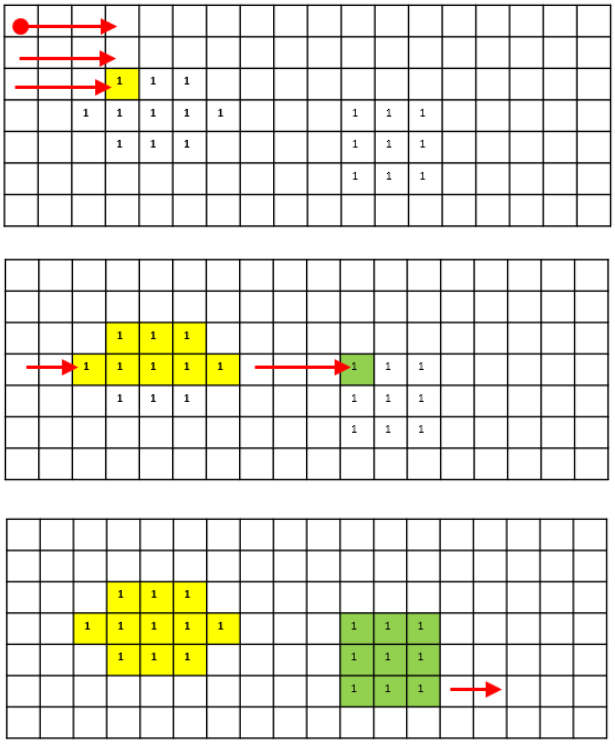
We have added some serious Gaussian noise to the original coin image which now appears in Fig.15 where the additional noise pixels have created an image whose histogram does not have a valley; it is *unimodal* rather than the *bimodal* histogram we want. Clearly it is impossible to assign a threshold to this image. So how do we proceed? Drawing on our understanding of smoothing noisy images, let's apply a 5x5 average smoothing kernel. The smoothed image and its histogram, and the results of Otsu's algorithm are shown in Fig.16.

The final segmentation result is almost perfect, and Otsu's algorithm has performed exceedingly well since the histogram is quite complicated having at least two troughs. The effect of smoothing has produced a relatively larger number of brighter pixels (the coins) even though there is still a huge number of pixels in the range 160-200.

Labelling and Analysing Objects

This is quite straightforward and is explained in the diagram below. We start at the top left pixel in the image and scan across the columns, and down the rows. If we find an object pixel (value = 1) then we increment our label, and label that pixel; this is shown as yellow in the diagram. Then we continue scanning, and if we find an object pixel neighbouring the one we have labelled, then we label that second pixel with the same label. You can see this in the first

diagram where one pixel is labelled yellow, and the next one to the right will also be labelled yellow.



In the second diagram, the next row of the first object is complete, then we hit another object pixel. This is no connected to the first labelled object, so we assign a new label, in this case green. Finally, all objects are labelled. Once they are labelled, we can return to this list of labelled objects and do some analysis, e.g., we can count the pixels to get their area, or find the object width and height or shape.