Chapter 6

Digital images

The theory on wavelets has been presented as a one-dimensional theory upto now. Images, however, are two-dimensional by nature. This poses another challenge, contrary to the case for sound. In the next chapter we will establish the mathematics to handle this, but first we will present some basics on images. Images are a very important type of digital media, and we will go through how they can be represented and manipulated with simple mathematics. This is useful general knowledge for anyone who has a digital camera and a computer, but for many scientists, it is an essential tool. In astrophysics, data from both satellites and distant stars and galaxies is collected in the form of images, and information extracted from the images with advanced image processing techniques. Medical imaging makes it possible to gather different kinds of information in the form of images, even from the inside of the body. By analysing these images it is possible to discover tumours and other disorders.

6.1 What is an image?

Before we do computations with images, it is helpful to be clear about what an image really is. Images cannot be perceived unless there is some light present, so we first review superficially what light is.

6.1.1 Light

Fact 6.1 (What is light?)**.** Light is electromagnetic radiation with wavelengths in the range 400–700 nm (1 nm is 10^{-9} m): Violet has wavelength 400 nm and red has wavelength 700 nm. White light contains roughly equal amounts of all wave lengths.

Other examples of electromagnetic radiation are gamma radiation, ultraviolet and infrared radiation and radio waves, and all electromagnetic radiation travel at the speed of light (3×10^8 m/s). Electromagnetic radiation consists of waves

and may be reflected and refracted, just like sound waves (but sound waves are not electromagnetic waves).

We can only see objects that emit light, and there are two ways that this can happen. The object can emit light itself, like a lamp or a computer monitor, or it reflects light that falls on it. An object that reflects light usually absorbs light as well. If we perceive the object as red it means that the object absorbs all light except red, which is reflected. An object that emits light is different; if it is to be perceived as being red it must emit only red light.

6.1.2 Digital output media

Our focus will be on objects that emit light, for example a computer display. A computer monitor consists of a rectangular array of small dots which emit light. In most technologies, each dot is really three smaller dots, and each of these smaller dots emit red, green and blue light. If the amounts of red, green and blue is varied, our brain merges the light from the three small light sources and perceives light of different colours. In this way the colour at each set of three dots can be controlled, and a colour image can be built from the total number of dots.

It is important to realise that it is possible to generate most, but not all, colours by mixing red, green and blue. In addition, different computer monitors use slightly different red, green and blue colours, and unless this is taken into consideration, colours will look different on the two monitors. This also means that some colours that can be displayed on one monitor may not be displayable on a different monitor.

Printers use the same principle of building an image from small dots. On most printers however, the small dots do not consist of smaller dots of different colours. Instead as many as 7–8 different inks (or similar substances) are mixed to the right colour. This makes it possible to produce a wide range of colours, but not all, and the problem of matching a colour from another device like a monitor is at least as difficult as matching different colours across different monitors.

Video projectors builds an image that is projected onto a wall. The final image is therefore a reflected image and it is important that the surface is white so that it reflects all colours equally.

The quality of a device is closely linked to the density of the dots.

Fact 6.2 (Resolution)**.** The resolution of a medium is the number of dots per inch (dpi). The number of dots per inch for monitors is usually in the range 70–120, while for printers it is in the range 150–4800 dpi. The horizontal and vertical densities may be different. On a monitor the dots are usually referred to as *pixels* (picture elements).

6.1.3 Digital input media

The two most common ways to acquire digital images is with a digital camera or a scanner. A scanner essentially takes a photo of a document in the form of a rectangular array of (possibly coloured) dots. As for printers, an important measure of quality is the number of dots per inch.

Fact 6.3. The resolution of a scanner usually varies in the range 75 dpi to 9600 dpi, and the colour is represented with up to 48 bits per dot.

For digital cameras it does not make sense to measure the resolution in dots per inch, as this depends on how the image is printed (its size). Instead the resolution is measured in the number of dots recorded.

Fact 6.4. The number of pixels recorded by a digital camera usually varies in the range 320×240 to 6000×4000 with 24 bits of colour information per pixel. The total number of pixels varies in the range 76 800 to 24 000 000 (0.077 megapixels to 24 megapixels).

For scanners and cameras it is easy to think that the more dots (pixels), the better the quality. Although there is some truth to this, there are many other factors that influence the quality. The main problem is that the measured colour information is very easily polluted by noise. And of course high resolution also means that the resulting files become very big; an uncompressed 6000×4000 image produces a 72 MB file. The advantage of high resolution is that you can magnify the image considerably and still maintain reasonable quality.

6.1.4 Definition of digital image

We have already talked about digital images, but we have not yet been precise about what it is. From a mathematical point of view, an image is quite simple.

Fact 6.5 (Digital image)**.** A digital image P is a rectangular array of *intensity values* $\{p_{i,j}\}_{i,j=1}^{m,n}$. For grey-level images, the value $p_{i,j}$ is a single number, while for colour images each $p_{i,j}$ is a vector of three or more values. If the image is recorded in the rgb-model, each $p_{i,j}$ is a vector of three values,

$$p_{i,j} = (r_{i,j}, g_{i,j}, b_{i,j}),$$

that denote the amount of red, green and blue at the point (i, j).

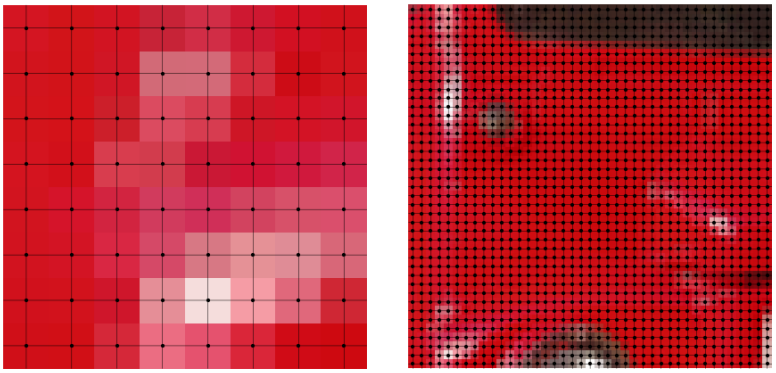
The value $p_{i,j}$ gives the colour information at the point (i, j). It is important to remember that there are many formats for this. The simplest case is plain black and white images in which case $p_{i,j}$ is either 0 or 1. For grey-level images the intensities are usually integers in the range 0–255. However, we will assume that the intensities vary in the interval $[0, 1]$, as this sometimes simplifies the form of some mathematical functions. For colour images there are

(a)

(b)

(c)

Figure 6.1: Different version of the same image; black and white (a), grey-level (b), and colour (c).



(a) (b)

Figure 6.2: Two excerpt of the colour image in figure 6.1. The dots indicate the position of the points (i, j).

many different formats, but we will just consider the rgb-format mentioned in the fact box. Usually the three components are given as integers in the range 0–255, but as for grey-level images, we will assume that they are real numbers in the interval $[0, 1]$ (the conversion between the two ranges is straightforward, see section 6.11 below). Figure 6.1 shows an image in different formats.

Fact 6.6. In these notes the intensity values $p_{i,j}$ are assumed to be real numbers in the interval $[0, 1]$. For colour images, each of the red, green, and blue intensity values are assumed to be real numbers in $[0, 1]$.

If we magnify a small part of the colour image in figure 6.1, we obtain the image in figure 6.2 (the black lines and dots have been added). A we can see, the pixels have been magnified to big squares. This is a standard representation used by many programs — the actual shape of the pixels will depend on the output medium. Nevertheless, we will consider the pixels to be square, with integer coordinates at their centres, as indicated by the grids in figure 6.2.

Fact 6.7 (Shape of pixel)**.** The pixels of an image are assumed to be square with sides of length one, with the pixel with value $p_{i,j}$ centred at the point (i, j).

6.1.5 Images as surfaces

Recall from your previous calculus courses that a function $f : \mathbb{R}^2 \mapsto \mathbb{R}$ can be visualised as a surface in space. A grey-level image is almost on this form. If we define the set of integer pairs by

$$\mathbb{Z}_{m,n} = \big\{(i, j) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\big\},$$

Figure 6.3: The grey-level image in figure 6.1 plotted as a surface. The height above the (x, y)-plane is given by the intensity value.

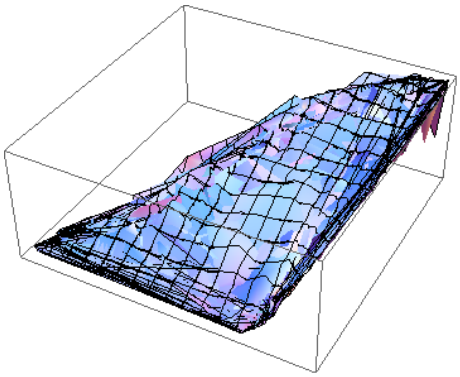


Figure 6.4: A colour image viewed as a parametric surface in space.

we can consider a grey-level image as a function $P : \mathbb{Z}_{m,n} \mapsto [0,1]$. In other words, we may consider an image to be a sampled version of a surface with the intensity value denoting the height above the (x, y)-plane, see figure 6.3.

Fact 6.8 (Grey-level image as a surface)**.** Let $P = (p)_{i,j=1}^{m,n}$ be a grey-level image. Then P can be considered a sampled version of the piecewise constant surface

$$F_P : [1/2, m + 1/2] \times [1/2, n + 1/2] \mapsto [0, 1]$$

which has the constant value $p_{i,j}$ in the square (pixel)

$$[i - 1/2, i + 1/2] \times [j - 1/2, j + 1/2]$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

What about a colour image P? Then each $p_{i,j} = (r_{i,j}, g_{i,j}, b_{i,j})$ is a triple of numbers so we have a mapping

$$P : \mathbb{Z}_{m,n} \mapsto \mathbb{R}^3.$$

If we examine this expression, we see that this corresponds to a sampled version of a parametric surface if we consider the colour values $(r_{i,j}, g_{i,j}, b_{i,j})$ to be x-, y-, and z-coordinates. This may be useful for computations in certain settings, but visually it does not make much sense, see figure 6.4

6.2 Operations on images

Images are two-dimensional arrays of numbers, contrary to the sound signals we considered in the previous section. In this respect it is quite obvious that we can manipulate an image by performing mathematical operations on the numbers. In this section we will consider some of the simpler operations. In later sections

we will go through more advanced operations, and explain how the theory for these can be generalized from the corresponding theory for one-dimensional (sound) signals (which we wil go through first).

In order to perform these operations, we need to be able to use images with a programming environment such as MATLAB.

6.2.1 Images and MATLAB

An image can also be thought of as a matrix, by associating each pixel with an element in a matrix. The matrix indices thus correspond to positions in the pixel grid. Black and white images correspond to matrices where the elements are natural numbers between 0 and 255. To store a colour image, we need 3 matrices, one for each colour component. This enables us to use linear algebra packages, such as MATLAB, on order to work with images. After we now have made the connection with matrices, we can create images from mathematical formulas, just as we could with sound in the previuos sections. But what we also need before we go through operations on images, is, as in the sections on sound, means of reading an image from a file so that its contents are accessible as a matrix, and write images represented by a matrix which we have constructed ourself to file. Reading a function from file can be done with help of the function `imread`. If we write

```
A = double(imread('filename.fmt','fmt'));
```

the image with the given path and format is read, and stored in the matrix `A`. 'fmt' can be 'jpg','tif', 'gif', 'png',... You should consult the MATLAB help pages to see which formats are supported. After the call to `imread`, we have a matrix where the entries represent the pixel values, and of integer data type (more precisely, the data type `uint8` in Matlab). To perform operations on the image, we must first convert the entries to the data type `double`. This is done with a call to the Matlab function `double`. Similarly, the function `imwrite` can be used to write the image represented by a matrix to file. If we write

```
imwrite(uint8(A), 'filename.fmt','fmt')
```

the image represented by the matrix `A` is written to the given path, in the given format. Before the image is written to file, you see that we have converted the matrix values back to the integer data type with the help of the function `uint8`. In other words: `imread` and `imwrite` both assume integer matrix entries, while operations on matrices assume double matrix entries. If you want to print images you have created yourself, you can use this function first to write the image to a file, and then send that file to the printer using another program. Finally, we need an alternative to playing a sound, namely displaying an image. The function `imageview(A)` displays the matrix `A` as an image in a separate window. Unfortunately, you can't print the image from the menus in this window.

The following examples go through some much used operations on images.

Example 6.9 (Normalising the intensities)**.** We have assumed that the intensities all lie in the interval $[0, 1]$, but as we noted, many formats in fact use integer values in the range 0–255. And as we perform computations with the intensities, we quickly end up with intensities outside $[0, 1]$ even if we start out with intensities within this interval. We therefore need to be able to *normalise* the intensities. This we can do with the simple linear function

$$g(x) = \frac{x - a}{b - a}, \quad a < b,$$

which maps the interval $[a, b]$ to $[0, 1]$. A simple case is mapping $[0, 255]$ to $[0, 1]$ which we accomplish with the scaling $g(x) = x/255$. More generally, we typically perform computations that result in intensities outside the interval $[0, 1]$. We can then compute the minimum and maximum intensities $p_{\min}$ and $p_{\max}$ and map the interval $[p_{\min}, p_{\max}]$ back to $[0, 1]$. Below we have shown a function `mapto01` which achieves this task.

```
function newimg=mapto01(img)
  minval = min(min(img));
  maxval = max(max(img));
  newimg = (img - minval)/(maxval-minval);
```

Several examples of using this function will be shown below.

Example 6.10 (Extracting the different colours)**.** If we have a colour image $P = (r_{i,j}, g_{i,j}, b_{i,j})_{i,j=1}^{m,n}$ it is often useful to manipulate the three colour components separately as the three images

$$P_r = (r_{i,j})_{i,j=1}^{m,n}, \quad P_r = (g_{i,j})_{i,j=1}^{m,n}, \quad P_r = (b_{i,j})_{i,j=1}^{m,n}.$$

As an example, let us first see how we can produce three separate images, showing the R,G, and B colour components, respectively. Let us take the image `bus-small-rgb.png` used in Figure 6.1. When the image is read, three matrices are returned, one for each colour component, and we can generate new files for the different colour components with the following code:

```
img=double(imread('../images/bus-small-rgb.png','png'));
newimg=zeros(size(img));
newimg(:,:,1)=img(:,:,1);
imwrite(uint8(newimg),'../images/gr.png','png');
newimg=zeros(size(img));
newimg(:,:,2)=img(:,:,2);
imwrite(uint8(newimg),'../images/ggg.png','png');
newimg=zeros(size(img));
newimg(:,:,3)=img(:,:,3);
imwrite(uint8(newimg),'../images/gb.png','png');
```

The resulting image files are shown in Figure 6.5.

(a) (b) (c)

Figure 6.5: The red (a), green (b), and blue (c) components of the colour image in Figure 6.1.

Example 6.11 (Converting from colour to grey-level)**.** If we have a colour image we can convert it to a grey-level image. This means that at each point in the image we have to replace the three colour values (r, g, b) by a single value p that will represent the grey level. If we want the grey-level image to be a reasonable representation of the colour image, the value p should somehow reflect the intensity of the image at the point. There are several ways to do this.

It is not unreasonable to use the largest of the three colour components as a measure of the intensity, i.e, to set $p = \max(r, g, b)$. The result of this can be seen in Figure 6.6(a).

An alternative is to use the sum of the three values as a measure of the total intensity at the point. This corresponds to setting $p = r + g + b$. Here we have to be a bit careful with a subtle point. We have required each of the r, g and b values to lie in the range $[0, 1]$, but their sum may of course become as large as 3. We also require our grey-level values to lie in the range $[0, 1]$ so after having computed all the sums we must normalise as explained above. The result can be seen in Figure 6.6(b).

A third possibility is to think of the intensity of (r, g, b) as the length of the colour vector, in analogy with points in space, and set $p = \sqrt{r^2 + g^2 + b^2}$. Again, we may end up with values in the range $[0, \sqrt{3}]$ so we have to normalise like we did in the second case. The result is shown in Figure 6.6(c).

Let us sum this up as an algorithm.

This can be implemented by using most of the code from the previous example, and replacing with the lines

```
newimg1=max(img,[],3);
newvals=img(:,:,1)+img(:,:,2)+img(:,:,3);
newimg2=newvals/max(max(newvals))*255;
newvals=sqrt(img(:,:,1).^2+img(:,:,2).^2+img(:,:,3).^2);
```

A colour image $P = (r_{i,j}, g_{i,j}, b_{i,j})_{i,j=1}^{m,n}$ can be converted to a grey level image $Q = (q_{i,j})_{i,j=1}^{m,n}$ by one of the following three operations:

1. Set $q_{i,j} = \max(r_{i,j}, g_{i,j}, b_{i,j})$ for all i and j.
2. (a) Compute $\hat{q}_{i,j} = r_{i,j} + g_{i,j} + b_{i,j}$ for all i and j.
   (b) Transform all the values to the interval $[0, 1]$ by setting
   $$q_{i,j} = \frac{\hat{q}_{i,j}}{\max_{k,l} \hat{q}_{k,l}}.$$
3. (a) Compute $\hat{q}_{i,j} = \sqrt{r_{i,j}^2 + g_{i,j}^2 + b_{i,j}^2}$ for all i and j.
   (b) Transform all the values to the interval $[0, 1]$ by setting
   $$q_{i,j} = \frac{\hat{q}_{i,j}}{\max_{k,l} \hat{q}_{k,l}}.$$

```
newimg3=newvals/max(max(newvals))*255;
```

respectively. In practice one of the last two methods are usually preferred, perhaps with a preference for the last method, but the actual choice depends on the application. These resuliting images are visualised as grey-level images in Figure 6.5.

Example 6.12 (Computing the negative image)**.** In film-based photography a negative image was obtained when the film was developed, and then a positive image was created from the negative. We can easily simulate this and compute a negative digital image.

Suppose we have a grey-level image $P = (p_{i,j})_{i,j=1}^{m,n}$ with intensity values in the interval $[0, 1]$. Here intensity value 0 corresponds to black and 1 corresponds to white. To obtain the negative image we just have to replace an intensity p by its 'mirror value' $1 - p$.

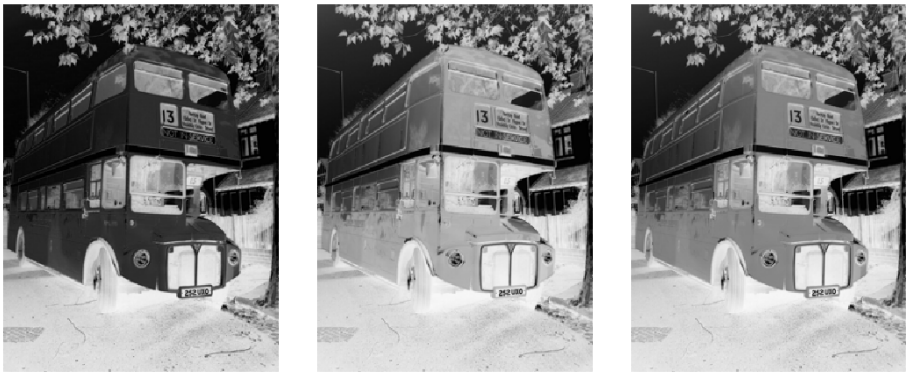
Fact 6.13 (Negative image)**.** Suppose the grey-level image $P = (p_{i,j})_{i,j=1}^{m,n}$ is given, with intensity values in the interval $[0, 1]$. The negative image $Q = (q_{i,j})_{i,j=1}^{m,n}$ has intensity values given by $q_{i,j} = 1 - p_{i,j}$ for all i and j.

This is also easily translated to code as above. The resulting image is shown in Figure 6.7.

Example 6.14 (Increasing the contrast)**.** A common problem with images is that the contrast often is not good enough. This typically means that a large proportion of the grey values are concentrated in a rather small subinterval of

(a) (b) (c)

Figure 6.6: Alternative ways to convert the colour image in Figure 6.1 to a grey level image. In (a) each colour triple has been replaced by its maximum, in (b) each colour triple has been replaced by its sum and the result mapped to $[0, 1]$, while in (c) each triple has been replaced by its length and the result mapped to $[0, 1]$.



(a) (b) (c)

Figure 6.7: The negative versions of the corresponding images in figure 6.6.

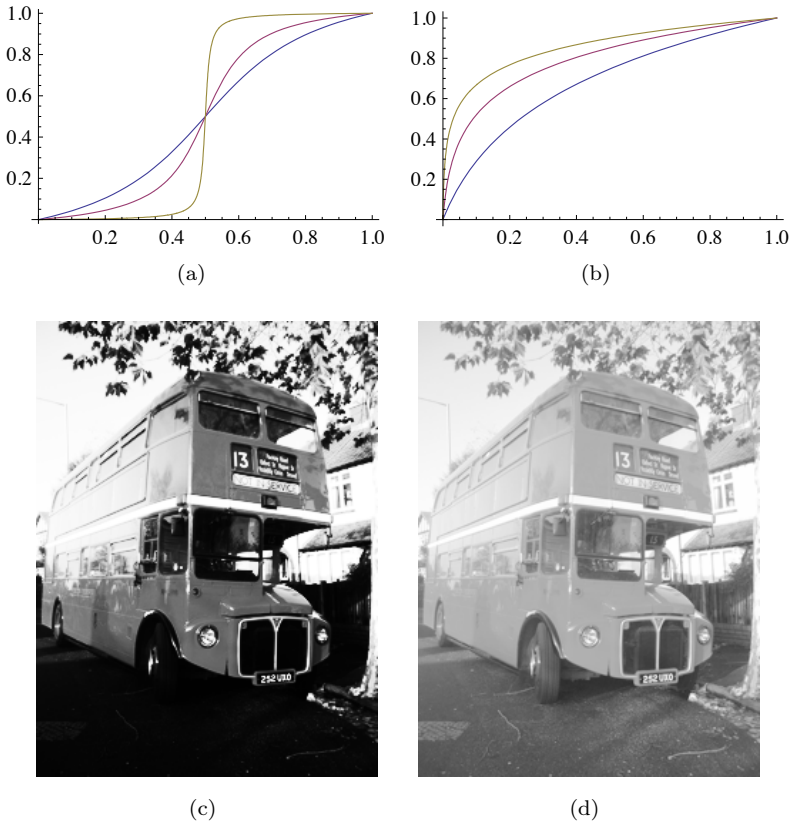


Figure 6.8: The plots in (a) and (b) show some functions that can be used to improve the contrast of an image. In (c) the middle function in (a) has been applied to the intensity values of the image in figure 6.6c, while in (d) the middle function in (b) has been applied to the same image.

$[0,1]$. The obvious solution to this problem is to somehow spread out the values. This can be accomplished by applying a function f to the intensity values, i.e., new intensity values are computed by the formula

$$\hat{p}_{i,j} = f(p_{i,j})$$

for all i and j. If we choose f so that its derivative is large in the area where many intensity values are concentrated, we obtain the desired effect.

Figure 6.8 shows some examples. The functions in the left plot have quite large derivatives near $x = 0.5$ and will therefore increase the contrast in images with a concentration of intensities with value around 0.5. The functions are all on the form

$$f_n(x) = \frac{\arctan\big(n(x-1/2)\big)}{2\arctan(n/2)} + \frac{1}{2}. \tag{6.1}$$

For any $n \neq 0$ these functions satisfy the conditions $f_n(0) = 0$ and $f_n(1) = 1$. The three functions in figure 6.8a correspond to $n = 4$, 10, and 100.

Functions of the kind shown in figure 6.8b have a large derivative near $x = 0$ and will therefore increase the contrast in an image with a large proportion of small intensity values, i.e., very dark images. The functions are given by

$$g_\epsilon(x) = \frac{\ln(x+\epsilon) - \ln\epsilon}{\ln(1+\epsilon) - \ln\epsilon}, \tag{6.2}$$

and the ones shown in the plot correspond to $\epsilon = 0.1$, 0.01, and 0.001.

In figure 6.8c the middle function in (a) has been applied to the image in figure 6.6c. Since the image was quite well balanced, this has made the dark areas too dark and the bright areas too bright. In figure 6.8d the function in (b) has been applied to the same image. This has made the image as a whole too bright, but has brought out the details of the road which was very dark in the original.

Observation 6.15. Suppose a large proportion of the intensity values $p_{i,j}$ of a grey-level image P lie in a subinterval I of $[0,1]$. Then the contrast of the image can be improved by computing new intensities $\hat{p}_{i,j} = f(p_{,j})$ where f is a function with a large derivative in the interval I.

Increasing the contrast is easy to implement. The following function has been used to generate the image in Figure 6.8(d):

```
function newimg=contrastadjust(img)
  epsilon = 0.001;  % Try also 0.1, 0.01, 0.001
  newimg = img/255; % Maps the pixel values to [0,1]
  newimg = (log(newimg+epsilon) - log(epsilon))/...
           (log(1+epsilon)-log(epsilon));
  newimg = newimg*255; % Maps the values back to [0,255]
```

We will see more examples of how the contrast in an image can be enhanced when we try to detect edges below.

(a) (b) (c)

Figure 6.9: The images in (b) and (c) show the effect of smoothing the image in (a).

Example 6.16 (Smoothing an image)**.** When we considered filtering of digital sound, we observed that replacing each sample of a sound by an average of the sample and its neighbours dampened the high frequencies of the sound. We can do a similar operation on images.

Consider the array of numbers given by

$$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}. \tag{6.3}$$

We can smooth an image with this array by placing the centre of the array on a pixel, multiplying the pixel and its neighbours by the corresponding weights, summing up and dividing by the total sum of the weights. More precisely, we would compute the new pixels by

$$\hat{p}_{i,j} = \frac{1}{16}\big(4p_{i,j} + 2(p_{i,j-1} + p_{i-1,j} + p_{i+1,j} + p_{i,j+1}) + p_{i-1,j-1} + p_{i+1,j-1} + p_{i-1,j+1} + p_{i+1,j+1}\big).$$

Since the weights sum to one, the new intensity value $\hat{p}_{i,j}$ is a weighted average of the intensity values on the right. The array of numbers in (6.3) is in fact an example of a computational molecule. As in the section on sound, we could have used equal weights for all pixels, but it seems reasonable that the weight of a pixel should be larger the closer it is to the centre pixel. For the onedimensional case on sound, we used the values of Pascal's triangle here, since these weights are known to give a very good smoothing effect. We will return to how we can generalize the use of Pascal's triangle to obtain computational molecules for use in images.

A larger filter is given by the array

$$\frac{1}{4096}\begin{pmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{pmatrix}. \tag{6.4}$$

These numbers are taken from row six of Pascal's triangle. More precisely, the value in row k and column l is given by the product $\binom{6}{k}\binom{6}{l}$. The scaling $1/4096$ comes from the fact that the sum of all the numbers in the table is $2^{6+6} = 4096$.

The result of applying the two filters in (6.3) and (6.4) to our greyscale-image is shown in Figure 6.9(b) and -(c) respectively. The smoothing effect is clearly visible.

Observation 6.17. An image P can be smoothed out by replacing the intensity value at each pixel by a weighted average of the intensity at the pixel and the intensity of its neighbours.

It is straightforward to write a function which performs smoothing. Assume that the image is stored as the matrix `img`, and the computational molecule is stored as the matrix `compmolecule`. The following function will return the smoothed image:

```
function newimg=smooth(img,compmolecule)
  [m,n]=size(img);
  [k,k1] = size(compmolecule); % We need k==k1, and odd
  sc = (k+1)/2;
  for m1=1:m
    for n1=1:n
      slidingwdw = zeros(k,k);
        % slidingwdw is the part of the picture which
        % compmolecule is applied to pixel (m1,n1)
      slidingwdw(max(sc+1-m1,1):min(sc+m-m1,2*sc-1) , ...
                 max(sc+1-n1,1):min(sc+n-n1,2*sc-1)) = ...
      img(max(1,m1-(sc-1)):min(m,m1+(sc-1)) , ...
          max(1,n1-(sc-1)):min(n,n1+(sc-1)));
      newimg(m1,n1) = sum(sum(compmolecule .* slidingwdw));
    end
  end
```

What makes this code difficult to write is the fact that the computational molecule may extend outside the borders of the image, when we are close to these borders. With this function, the first smoothing above can be performed by writing

```
smooth(img,(1/16)*[ 1 2 1; 2 4 2; 1 2 1]);
```

Example 6.18 (Detecting edges)**.** The final operation on images we are going to consider is edge detection. An edge in an image is characterised by a large change in intensity values over a small distance in the image. For a continuous function this corresponds to a large derivative. An image is only defined at isolated points, so we cannot compute derivatives, but we have a perfect situation for applying numerical differentiation. Since a grey-level image is a scalar function of two variables, numerical differentiation techniques can be applied.

Partial derivative in x-direction. Let us first consider computation of the partial derivative $\partial P/\partial x$ at all points in the image. We use the familiar approximation

$$\frac{\partial P}{\partial x}(i,j) = \frac{p_{i+1,j} - p_{i-1,j}}{2}, \tag{6.5}$$

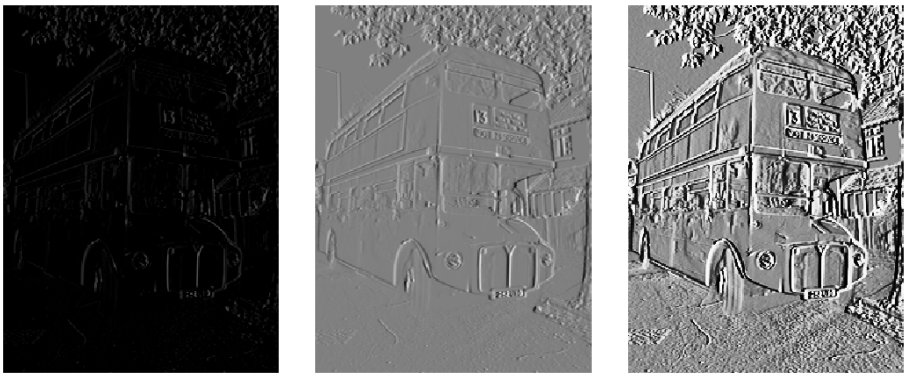
where we have used the convention $h = 1$ which means that the derivative is measured in terms of 'intensity per pixel'. We can run through all the pixels in the image and compute this partial derivative, but have to be careful for $i = 1$ and $i = m$ where the formula refers to non-existing pixels. We will adapt the simple convention of assuming that all pixels outside the image have intensity 0. The result is shown in figure 6.10a.

This image is not very helpful since it is almost completely black. The reason for this is that many of the intensities are in fact negative, and these are just displayed as black. More specifically, the intensities turn out to vary in the interval $[-0.424, 0.418]$. We therefore normalise and map all intensities to $[0, 1]$. .. The result of this is shown in (b). The predominant colour of this image is an average grey, i.e, an intensity of about 0.5. To get more detail in the image we therefore try to increase the contrast by applying the function f_{50} in equation 6.1 to each intensity value. The result is shown in figure 6.10c which does indeed show more detail.

It is important to understand the colours in these images. We have computed the derivative in the x-direction, and we recall that the computed values varied in the interval $[-0.424, 0.418]$. The negative value corresponds to the largest average decrease in intensity from a pixel $p_{i-1,j}$ to a pixel $p_{i+1,j}$. The positive value on the other hand corresponds to the largest average increase in intensity. A value of 0 in figure 6.10a corresponds to no change in intensity between the two pixels.

When the values are mapped to the interval $[0, 1]$ in figure 6.10b, the small values are mapped to something close to 0 (almost black), the maximal values are mapped to something close to 1 (almost white), and the values near 0 are mapped to something close to 0.5 (grey). In figure 6.10c these values have just been emphasised even more.

Figure 6.10c tells us that in large parts of the image there is very little variation in the intensity. However, there are some small areas where the intensity changes quite abruptly, and if you look carefully you will notice that in these



(a) (b) (c)

Figure 6.10: The image in (a) shows the partial derivative in the x-direction for the image in 6.6. In (b) the intensities in (a) have been normalised to $[0, 1]$ and in (c) the contrast as been enhanced with the function f_{50}, equation 6.1.

areas there is typically both black and white pixels close together, like down the vertical front corner of the bus. This will happen when there is a stripe of bright or dark pixels that cut through an area of otherwise quite uniform intensity.

Since we display the derivative as a new image, the denominator is actually not so important as it just corresponds to a constant scaling of all the pixels; when we normalise the intensities to the interval $[0, 1]$ this factor cancels out.

We sum up the computation of the partial derivative by giving its computational molecule.

Observation 6.19. Let $P = (p_{i,j})_{i,j=1}^{m,n}$ be a given image. The partial derivative $\partial P/\partial x$ of the image can be computed with the computational molecule

$$\frac{1}{2}\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}. \tag{6.6}$$

As we remarked above, the factor $1/2$ can usually be ignored. We have included the two rows of 0s just to make it clear how the computational molecule is to be interpreted; it is obviously not necessary to multiply by 0.

Partial derivative in y-direction. The partial derivative $\partial P/\partial y$ can be computed analogously to $\partial P/\partial x$.

(a) (b) (c)

Figure 6.11: Computing the gradient. The image obtained from the computed gradient is shown in (a) and in (b) the numbers have been normalised. In (c) the contrast has been enhanced with a logarithmic function.

Observation 6.20. Let $P = (p_{i,j})_{i,j=1}^{m,n}$ be a given image. The partial derivative $\partial P/\partial y$ of the image can be computed with the computational molecule

$$\frac{1}{2}\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}. \tag{6.7}$$

The result is shown in figure 6.12b. The intensities have been normalised and the contrast enhanced by the function f_{50} in (6.1).

The gradient. The gradient of a scalar function is often used as a measure of the size of the first derivative. The gradient is defined by the vector

$$\nabla P = \left(\frac{\partial P}{\partial x}, \frac{\partial P}{\partial y}\right),$$

so its length is given by

$$|\nabla P| = \sqrt{\left(\frac{\partial P}{\partial x}\right)^2 + \left(\frac{\partial P}{\partial y}\right)^2}.$$

When the two first derivatives have been computed it is a simple matter to compute the gradient vector and its length; the resulting is shown as an image in figure 6.11c.

(a) (b) (c)

Figure 6.12: The first-order partial derivatives in the x-direction (a) and y-direction (b), and the length of the gradient (c). In all images, the computed numbers have been normalised and the contrast enhanced.

The image of the gradient looks quite different from the images of the two partial derivatives. The reason is that the numbers that represent the length of the gradient are (square roots of) sums of squares of numbers. This means that the parts of the image that have virtually constant intensity (partial derivatives close to 0) are coloured black. In the images of the partial derivatives these values ended up in the middle of the range of intensity values, with a final colour of grey, since there were both positive and negative values.

Figure 6.11a shows the computed values of the gradient. Although it is possible that the length of the gradient could become larger than 1, the maximum value in this case is about 0.876. By normalising the intensities we therefore increase the contrast slightly and obtain the image in figure 6.11b.

To enhance the contrast further we have to do something different from what was done in the other images since we now have a large number of intensities near 0. The solution is to apply a function like the ones shown in figure 6.8b to the intensities. If we use the function $g_{0.01}$ defined in equation(6.2) we obtain the image in figure 6.11c.

6.2.2 Comparing the first derivatives

Figure 6.12 shows the two first-order partial derivatives and the gradient. If we compare the two partial derivatives we see that the x-derivative seems to emphasise vertical edges while the y-derivative seems to emphasise horizontal edges. This is precisely what we must expect. The x-derivative is large when the difference between neighbouring pixels in the x-direction is large, which is the case across a vertical edge. The y-derivative enhances horizontal edges for a similar reason.

The gradient contains information about both derivatives and therefore em-

(a) (b) (c)

Figure 6.13: The second-order partial derivatives in the x-direction (a) and xy-direction (b), and the y-direction (c). In all images, the computed numbers have been normalised and the contrast enhanced.

phasises edges in all directions. It also gives a simpler image since the sign of the derivatives has been removed.

6.2.3 Second-order derivatives

To compute the three second order derivatives we apply the corresponding computational molecules which we already have described.

Observation 6.21 (Second order derivatives of an image)**.** The second order derivatives of an image P can be computed by applying the computational molecules

$$\frac{\partial^2 P}{\partial x^2}: \quad \begin{pmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{pmatrix}, \tag{6.8}$$

$$\frac{\partial^2 P}{\partial y \partial x}: \quad \frac{1}{4}\begin{pmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{pmatrix}, \tag{6.9}$$

$$\frac{\partial^2 P}{\partial y^2}: \quad \begin{pmatrix} 0 & -1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{pmatrix}. \tag{6.10}$$

With the information in observation 6.21 it is quite easy to compute the second-order derivatives, and the results are shown in figure 6.13. The computed derivatives were first normalised and then the contrast enhanced with the function f_{100} in each image, see equation 6.1.

As for the first derivatives, the xx-derivative seems to emphasise vertical edges and the yy-derivative horizontal edges. However, we also see that the second derivatives are more sensitive to noise in the image (the areas of grey are less uniform). The mixed derivative behaves a bit differently from the other two, and not surprisingly it seems to pick up both horizontal and vertical edges.

Exercises for section 6.2

Ex. 1 — Black and white images can be generated from greyscale images (with values between 0 and 255) by replacing each pixel value with the one of 0 and 255 which is closest. Use this strategy to generate the black and white image shown in Figure 6.1(a).

Ex. 2 — Generate the image in Figure 6.8(d) on your own by writing code which uses the function `contrastadjust`.

Ex. 3 — Let us also consider the second way we mentioned for increasing the contrast.

a. Write a function `contrastadjust2` which instead uses the function 6.1 to increase the contrast.

b. Generate the image in Figure 6.8(c) on your own by using your code from Exercise 2, and instead calling the function `contrastadjust2`.

Ex. 4 — In this exercise we will look at another function for increasing the contrast of a picture.

a. Show that the function $f : \mathbb{R} \to \mathbb{R}$ given by

$$f_n(x) = x^n,$$

for all n maps the interval $[0, 1] \to [0, 1]$, and that $f'(1) \to \infty$ as $n \to \infty$.

b. The color image `secret.jpg`,shown in Figure 6.14, contains some information that is nearly invisible to the naked eye on most computer monitors. Use the function $f(x)$, to reveal the secret message.
Hint: You will first need to convert the image to a greyscale image. You can then use the function `contrastadjust` as a starting point for your own program.

Ex. 5 — Generate the image in Figure 6.9(b) and -(c) by writing code which calls the function `smooth` with the appropriate computational molecules.

Figure 6.14: Secret message

Ex. 6 — Generate the image in Figure 6.10 by writing code in the same way. Also generate the images in figures 6.11, 6.12, and 6.13.

6.3 Adaptations to image processing

There are several extensions, additions and modifications to the theory we will present in the later chapters, which are needed in order for us to have a full image compression system: the wavelet transform, the DCT, and the DFT were addressed because these are linked to relevant mathematics, and because they are important ingredients in many standards for sound and images. In this section we will mention many other extensions which also are important.

6.3.1 Lossless coding

Somewhere in the image processing or sound processing pipeline we need a step which actually achieves compression of the data. We have not mentioned anything about this step, since the output from the wavelet transform or the DCT is simply another set of data of the same size, called the *transformed data*. What we need to do is to apply a coding algorithm to this data to achieve compression. This may be Huffman coding, arithmetic coding, or any other algorithm. These methods are applied to the transformed data, since the effect of the wavelet transform is that it exploits the data so that it can be represented with data with lower entropy, so that it can be compressed more efficiently with these techniques.

6.3.2 Quantization

Coding as we have learnt previously is a lossless operation. As we saw, for certain wavelets the transform can also be performed in a lossless manner. These wavelets are, however, quite restrictive, which is why there is some loss involved with most wavelets used in practical applications. When there is some loss

inherent in the transform, a quantization of the transformed data is also performed before the coding takes place. This quantization is typically done with a fixed number of bits, but may also be more advanced.

6.3.3 Preprocessing

Image compression as performed for certain image standards also often preprocess the pixel values before any transform is applied. The preprocessing may be centering the pixel values around a certain value, or extracting the different image components before they are processed separately.

6.3.4 Tiles, blocks, and error resilience

We have presented the wavelet transform as something which transforms the entire image. In practice this is not the case. The image is very often split into smaller parts, often called tiles. The tiles in an image are processed independently, so that errors whih occur within one tile do not affect the appearance of parts in the image which correspond to other tiles. This makes the image compression what we call *error-resilient*, to errors such as transmission errors. The second reason for splitting into tiles has to do with that it may be more efficient to perform many transforms on smaller parts, rather than one big transform for the entire image. Performing one big transform would force us to have a big part of the image in memory during each computation, as well as remove possibilities for parallel computing. Often the algorithm itself also requires more computations when the size is bigger. For some standards, tiles are split into even smaller parts, called blocks, where parts of the processing within each block also is performed independently. This makes the possibilities for parallel computing even bigger. As an example, we mentioned that the JPEG standard performs the two-dimensional DCT on blocks as small as size 8×8.

6.3.5 Metadata

An image standard also defines how to store metadata about an image, and what metadata is accepted, like resolution, time when the image was taken, or where the image was taken (GPS coordinates) and similar information. Metadata can also tell us how the colour in the image are represented. As we have already seen, in most colour images the colour of a pixel is represented in terms of the amount of red, green and blue or (r, g, b). But there are other possibilities as well: Instead of storing all 24 bits of colour information in cases where each of the three colour components needs 8 bits, it is common to create a table of 256 colours with which a given image could be represented quite well. Instead of storing the 24 bits, one then just stores a colour table in the metadata, and at each pixel, the eight bits corresponding to the correct entry in the table. This is usually referred to as eight-bit colour and the table is called a *look-up* table or *palette*. For large photographs, however, 256 colours is far from sufficient to obtain reasonable colour reproduction.

Metadata is usually stored in the beginning of the file, formatted in a very specific way.

6.4 Summary

We first discussed the basic question what an image is, and took a closer look at digital images. We went through several operations which give meaning for digital images, and showed how to implement these.