12

Images and Image Processing

Image processing is a computational process that transforms one or more input images into an output image. Image processing is frequently used to enhance an image for *human* viewing or interpretation, for example to improve contrast. Alternatively, and of more interest to robotics, it is the foundation for the process of feature extraction which will be discussed in much more detail in the next chapter.

An image is a rectangular array of picture elements (pixels) so we will use a MATLAB® matrix to represent an image in the workspace. This allows us to use MATLAB's powerful and efficient armory of matrix operators and functions.

We start in Sect. 12.1 by describing how to load images into MATLAB from sources such as files (images and movies), cameras and the internet. Next, in Sect. 12.2, we introduce image histograms which provide useful information about the distribution of pixel values. We then discuss various classes of image processing algorithms. These algorithms operate pixel-wise on a single image, a pair of images, or on local groups of pixels within an image and we refer to these as monadic, diadic, and spatial operations respectively. Monadic and diadic operations are covered in Sect. 12.3 and 12.4. Spatial operators are described in Sect. 12.5 and include operations such as smoothing, edge detection, and template matching. A closely related technique is shape-specific filtering or mathematical morphology and this is described in Sect. 12.6. Finally in Sect. 12.7 we discuss shape changing operations such as cropping, shrinking, expanding, as well as more complex operations such as rotation and generalized image warping.

Robots will always gather imperfect images of the world due to noise, shadows, reflections and uneven illumination. In this chapter we discuss some fundamental tools and "tricks of the trade" that can be applied to real-world images.

12.1 Obtaining an Image

Today digital images are ubiquitous since cameras are built into our digital devices and images cost almost nothing to create and share. We each have ever growing personal collections and access to massive online collections of digital images such as Google Images, Picasa or Flickr. We also have access to live image streams from other people's cameras – there are tens of thousands of webcams around the world capturing images and broadcasting them on the internet, as well images of Earth from space, the Moon and Mars.

12.1.1 Images from Files

We start with images stored in files since it is very likely that you already have lots of images stored on your computer. In this chapter we will work with some images provided with the Toolbox, but you can easily substitute your own images. We import an image into the MATLAB workspace using the Toolbox function `iread`

```
>> street = iread('street.png');
```

which returns a matrix

```
>> about(street)
street [uint8] : 851x1280 (1.1 MB)
```

that belongs to the class `uint8` – the elements of the matrix are unsigned 8-bit integers in the interval [0,255]. The elements are referred to as pixel values or grey values and are the gamma-encoded▶ luminance of that point in the original scene. For this 8-bit image the pixel values vary from 0 (darkest) to 255 (brightest). The image is shown in Fig. 12.1a.

Gamma encoding and decoding is discussed in Sect. 10.3.6. Use the `'gamma'` option for `iread` to perform gamma decoding and obtain pixel values proportional to scene luminance.

The matrix has 851 rows and 1 280 columns. We normally describe the dimensions of an image in terms of its width × height, so this would be a 1 280 × 851 pixel image.

> In Chap. 11 we wrote the coordinates of a pixel as (u, v) which are the horizontal and vertical coordinates respectively. In MATLAB this is the matrix element (v, u) – note the reversal of coordinates. Note also that the top-left pixel is (1, 1) in MATLAB not (0, 0).

For example the pixel at image coordinate (300, 200) is

```
>> street(200,300)
ans =
    42
```

which is quite dark – the pixel corresponds to a point in the closest doorway.

There are some subtleties when working with `uint8` values in MATLAB which we can demonstrate by defining two `uint8` values

```
>> a = uint8(100)
a =
   100
>> b = uint8(200)
b =
   200
```

Arithmetic on `uint8` values obeys the rules of `uint8` arithmetic

```
>> a+b
ans =
   255
>> a-b
ans =
     0
```

and values are clipped to the interval 0 to 255. For division
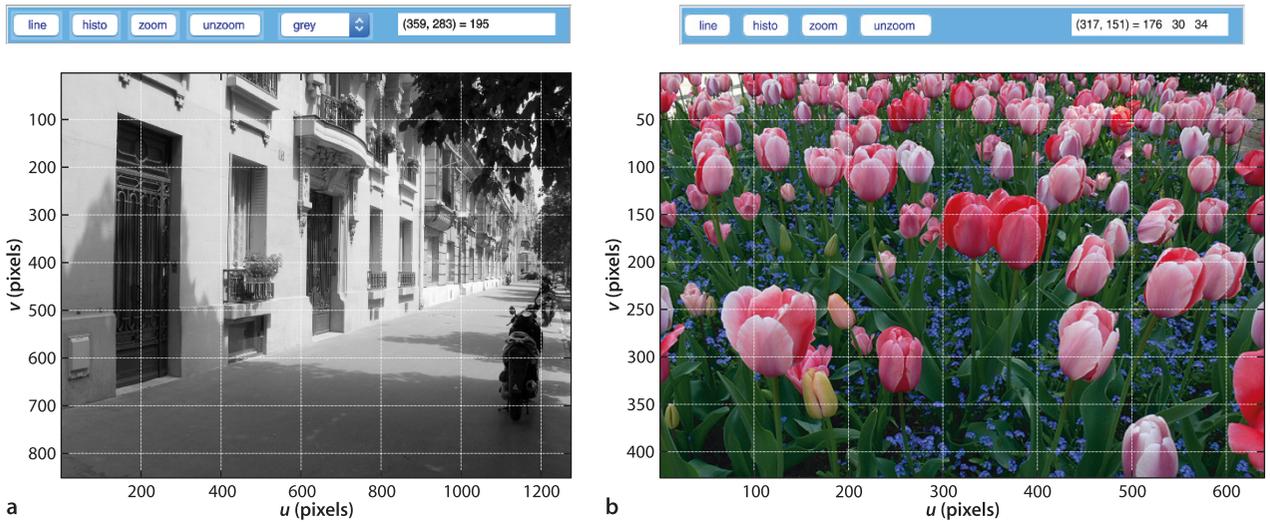
```
>> a/b
ans =
     1
```

the result has been rounded up to an integer value. For some image processing operations that we will consider later it is useful to consider the pixel values as floating-point

A very large number of **image file formats** have been developed and are comprehensively catalogued at http://en.wikipedia.org/wiki/Image_file_formats. The most popular is JPEG which is used for digital cameras and webcams. TIFF is common in many computer systems and often used for scanners. PNG and GIF are widely used on the web. The internal format of these files are complex but a large amount of good quality open-source software exists in a variety of languages to read and write such files. MATLAB is able to read many of these image file formats.

A much simpler set of formats, widely used on Linux systems, are PBM, PGM and PPM (generically PNM) which represent images without compression, and optionally as readable ASCII text. A host of open-source tools such as ImageMagick provide format conversions and image manipulation under Linux, MacOS X and Windows. (word map by tagxedo.com)

```
line   histo   zoom   unzoom   grey ⬍     (359, 283) = 195
```

```
line   histo   zoom   unzoom          (317, 151) = 176  30  34
```

a   *u* (pixels)

b   *u* (pixels)

Fig. 12.1. The `idisp` image browsing window. The top right shows the coordinate and value of the last pixel clicked on the image. The *buttons* at the top left allow the pixel values along a line to be plotted, a histogram to be displayed, or the image to be zoomed. **a** Greyscale image; **b** color image

Lossless means that the compressed image, when uncompressed, will be exactly the same as the original image.

The example images are kept within the `images` folder of the Machine Vision Toolbox distribution which is automatically searched by the `iread` function.

The colormap controls the mapping of pixel values to the displayed intensity or color.

numbers for which more familiar arithmetic rules apply. In this case each pixel is an 8-byte MATLAB double precision number in the range [0, 1] and we can specify this as an option when we load the image

```
>> streetd = iread('street.png', 'double');
>> about streetd
streetd [double] : 851x1280 (8.7 MB)
```

or by applying the function `idouble` to the integer image.

The image was read from a file called `street.png` which is in portable network graphics (PNG) format – a lossless compression format◄ widely used on the internet. The function `iread` searches for the image in the current folder, and then in each folder along your MATLAB path.◄ This particular image has no color, it is a greyscale or monochromatic image.

A tool that we will use a lot in this part of the book is `idisp`

```
>> idisp(street)
```

which displays the matrix as an image and allows interactive inspection of pixel values as shown in Fig. 12.1. Clicking on a pixel will display the pixel coordinate and its grey value – integer or floating point – in the top right of the window. The image can be zoomed (and unzoomed), we can display a histogram or the intensity profile along a line between any two selected points, or change the color map.◄ It has many options and these are described in the online documentation.

We can just as easily load a color image

```
>> flowers = iread('flowers8.png');
>> about(flowers)
flowers [uint8] : 426x640x3 (817920 bytes)
```

which is a $426 \times 640 \times 3$ matrix of `uint8` values as shown in Fig. 12.1b. We can think of this as a 426×640 matrix of RGB tristimulus values, each of which is a 3-vector. For example the pixel at (318, 276)

```
>> pix = flowers(276,318,:)
ans(:,:,1) =
    57
ans(:,:,2) =
    91
ans(:,:,3) =
   198
```

has a tristimulus value (57, 91, 198) but has been displayed by MATLAB in an unusual and noncompact manner. This is because the pixel value is
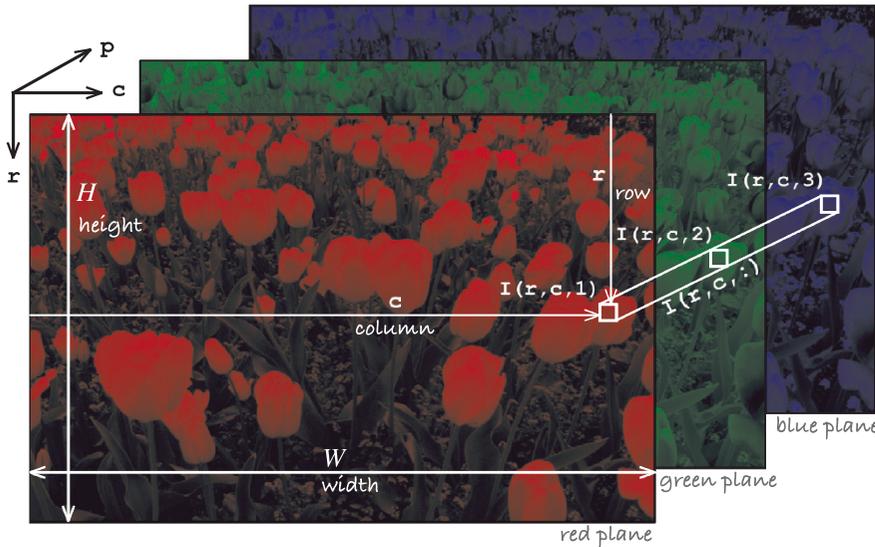
Fig. 12.2.
Color image shown as a 3-dimensional structure with dimensions: row, column, and color plane

```
>> about(pix)
pix [uint8] : 1x1x3 (3 bytes)
```

a $1 \times 1 \times 3$ matrix. The first two dimensions are called singleton dimensions and we can *squeeze* them out

```
>> squeeze(pix)'
ans =
    57    91    198
```

which results in a more familiar 3-vector. This pixel corresponds to one of the small blue flowers and has a large blue component. We can display the image and examine it interactively using `idisp` and clicking on a pixel will display its tristimulus value.

The tristimulus values are of type `uint8` in the range [0, 255] but the image can be converted to double precision values in the range [0, 1] using the `'double'` option to `iread` or by applying the function `idouble` to the integer color image, just as for a greyscale image.

The image is a matrix with three dimensions and the third dimension as shown in Fig. 12.2 is known as the color plane index. For example

```
>> idisp( flowers(:,:,1) )
```

would display the red color plane as a greyscale image that shows the red stimulus at each pixel. The index 2 or 3 would select the green or blue plane respectively.

The option `'grey'` ensures that a greyscale image is returned irrespective of whether or not the file contains a color image.▶ The option `'gamma'` performs gamma decoding and returns a linear image where the greylevels, or tristimulus values, are proportional to the luminance of the original scene.

The `iread` function can also accept a wildcard filename allowing it to load a sequence of files. For example

```
>> seq = iread('seq/*.png');
>> about(seq)
seq [uint8] : 512x512x9 (2.4 MB)
```

loads nine images in PNG format from the folder `seq`. The result is a $H \times W \times N$ matrix and the last index represents the image number within the sequence. That is `seq(:,:,k)` is the kth image in the sequence and is a 512×512 greyscale image. In terms of Fig. 12.2 the images in the sequence extend in the `p` direction. If the images were color then the result would be a $H \times W \times 3 \times N$ matrix where the last index represents the image number within the sequence, and the third index represents the color plane.

JPEG employs *lossy* compression to reduce the size of the file. Unlike normal file compression (eg. zip, rar, etc.) and decompression, the decompressed image isn't the same as the original image and this allows much greater levels of compression. JPEG compression exploits limitations of the human eye and discards information that won't be noticed such as very small color changes (which are perceived less accurately than small changes in brightness) and fine texture. It is very important to remember that JPEG is intended for compressing images that will be *viewed by humans*. The loss of color detail and fine texture may be problematic for computer algorithms that analyze images.

JPEG was designed to work well for natural scenes but it does not do so well on lettering and line drawings with high spatial-frequency content. The degree of loss can be varied by adjusting the so-called quality factor which allows a tradeoff between image quality and file size. JPEG can be used for greyscale or color images.

What is commonly referred to as a JPEG file, often with an extension of .jpg or .jpeg, is more correctly a JPEG JFIF file. JFIF is the format of the file that holds a JPEG-compressed image as well as metadata. EXIF file format (Exchangeable Image File Format) is a standard for camera related metadata such as camera settings, time, location and so on. This metadata can be retrieved as a second output argument to iread as a cell array, or by using a command-line utility such as exiftool (http://www.sno.phy. queensu.ca/~phil/exiftool). See the Independent JPEG group web site http://www.ijg.org for more details.

If iread is called with no arguments, a file browsing window pops up allowing navigation through the file system to find the image. The function also accepts a URL allowing it to load an image, but not a sequence, from the web. The function can read most common image file formats including JPEG, TIFF, GIF, PNG, PGM, PPM, PNM.

Many image file formats also contain rich metadata – data about the data in the file. The JPEG files generated by most digital cameras are particularly comprehensive and the metadata can be retrieved by providing a second output argument to iread

```
>> [im,md]=iread('church.jpg');
>> md
md = ...
                Width: 1280
               Height: 851
                 Make: 'Panasonic'
                Model: 'DMC-FZ30'
          Orientation: 1
        DigitalCamera: [1x1 struct]
```

and the DigitalCamera substructure has additional details about the camera settings for the particular image

```
>> md.DigitalCamera
ans = ...
           ExposureTime: 0.0025
                FNumber: 8
         ISOSpeedRatings: 80
                  Flash: 'Flash did not fire…'
            FocalLength: 7.4000
```

More details and options for iread are described in the online documentation.

12.1.2 Images from an Attached Camera

Most laptop computers today have a builtin camera for video conferencing. For computers without a builtin camera an external camera can be easily attached via a USB or FireWire connection. The means of accessing a camera is operating system specific. ◄

The Toolbox provides a simple interface to a camera for MacOS, Linux and Windows but more general support requires the Image Acquisition Toolbox.

A list of all attached cameras and their resolution can be obtained by

```
>> VideoCamera('?')
```

We open a particular camera

```
>> cam = VideoCamera('name')
```

which returns an instance of a VideoCamera object that is a subclass of the ImageSource class. If name is not provided the first camera found is used. The constructor accepts a number of additional arguments such as 'grey' which ensures that the returned image is greyscale irrespective of the camera type, 'gamma'

Photons to pixel values. A lot goes on inside a camera. Over a fixed time interval the number of photons falling on a photosite follows a Poisson distribution. The mean number of photons *and* the variance are proportional to the luminance – this variance appears as *shot noise* on the pixel value. A fraction of these photons are converted to electrons – this is the quantum efficiency of the sensor – and they accumulate in a charge well at the photosite. The number of photons captured is proportional to surface area but not all of a photosite is light sensitive due to the presence of transistors and other devices – the fraction of the photosite's area that is sensitive is called the fill factor and for CMOS sensors can be less than 50%, but this can be improved by fabricating microlenses above each photosite.

The charge well also accumulates thermally generated electrons, the dark current, which is proportional to temperature and is a source of noise – extreme low light cameras are cooled. Another source of noise is pixel nonuniformity due to adjacent pixels having a different gain or offset – uniform illumination therefore leads to pixels with different values which appears as additive noise. The charge well has a maximum capacity and with excessive illumination surplus electrons can overflow into adjacent charge wells leading to flaring and bleeding.

At the end of the exposure interval the accumulated charge (thermal- and photo-electrons) is read. For low-cost CMOS sensors the charge wells are connected sequentially via a switching network to one or more on-chip analog to digital converters. This results in a rolling shutter and for high speed relative motion this leads to tearing or jello effect as shown to the right. More expensive CMOS and CCD sensors have a global shutter – they make a temporary snapshot copy of the charge in a buffer which is then digitized sequentially.

The exposure on the sensor is

$$H = qL\frac{T}{N^2} \text{ lx s}$$

where L is scene luminance (in nit), T is exposure time, N is the f-number (inverse aperture diameter) and $q \approx 0.7$ is a function of

focal length, lens quality and vignetting. Exposure time T has an upper bound equal to the inverse frame rate. To avoid motion blur a short exposure time is needed, which leads to darker and noisier images.

The integer pixel value is

$$x = kH$$

where k is a gain related to the ISO setting[a] of the camera. To obtain an sRGB (see page 311) image with an average value of 118[b] the required exposure is
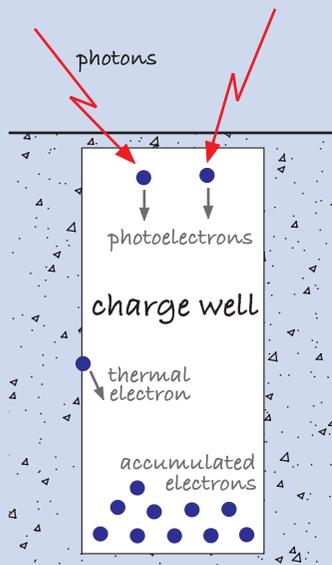
$$H = \frac{10}{S_{\text{SOS}}} \text{ lx s}$$

where S_{SOS} is the ISO rating – standard output sensitivity (SOS) – of the digital camera. Higher ISO increases image brightness by greater amplification of the measured charge but the various noise sources are also amplified leading to increased image noise which is manifested as graininess.

In photography the camera settings that control image brightness can be combined into an exposure value (EV)

$$\text{EV} = \log_2 \frac{N^2}{T}$$

and all combinations of f-number and shutter speed that have the same EV value yield the same exposure. This allows a tradeoff between aperture (depth of field) and exposure time (motion blur). For most low-end cameras the aperture is fixed and the camera controls exposure using T instead of relying on an expensive, and slow, mechanical aperture. A difference of 1 EV is a factor of two change in exposure which photographers refer to as a *stop*. Increasing EV results in a darker image – most DSLR cameras allow you to manually adjust EV relative to what the camera's lightmeter has determined.

---

[a] Which is backward compatible with historical scales (ASA, DIN, ISO) devised to reflect the sensitivity of chemical films for cameras – a higher number reflected a more sensitive or "faster" film.

[b] 18% saturation, middle grey, of 8-bit pixels with gamma of 2.2.

---

to apply gamma decoding, and `'framerate'` which sets the number of frames captured per second.

The dimensions of the image returned by the camera are given by the `size` method

```
>> cam.size()
```

and an image is obtained using the `grab` method

```
>> im = cam.grab();
```

which waits until the next frame becomes available.▶ With no output arguments the acquired image is displayed using `idisp`.

Since the frames are generated at a rate of R per second as specified by the `'framerate'` option, then the worst case wait is uniformly distributed in the interval $[0, 1/R]$. Ungrabbed images don't accumulate, they are discarded.

The **dynamic range** of a sensor is the ratio of its largest value to its smallest value. For images it is useful to express the $\log_2$ of this ratio which makes it equivalent to the photographic concepts of stops or exposure value. Each photosite contains a charge well in which photon-generated electrons are captured during the exposure period (see page 364). The charge well has a finite capacity before the photosite saturates and this defines the maximum value. The minimum number of electrons is not zero but a finite number of thermally generated electrons.

An 8-bit image has a dynamic range of around 8 stops, a high-end 10-bit camera has a range of 10 stops, and photographic film is perhaps in the range 10–12 stops but is quite nonlinear.

At a particular state of adaptation, the human eye has a range of 10 stops, but the total adaptation range is an impressive 20 stops. This is achieved by using the iris and slower (tens of minutes) chemical adaptation of the sensitivity of rod cells. Dark adaptation to low luminance is slow, whereas adaptation from dark to bright is faster but sometimes painful.

Video file formats. Just as for image files there are a large number of different file formats for videos. The most common formats are MPEG and AVI. It is important to distinguish between the format of the file (the *container*), technically AVI is a file format, and the type of compression (the *codec*) used on the images within the file.

MPEG and AVI format files can be converted to a sequence of frames as individual files using tools such as FFmpeg and `convert` from the ImageMagick suite. The individual frames can then be loaded individually into MATLAB for processing using `iread`. The Toolbox `Movie` class provides a more convenient way to read frames directly from common movie formats without having to first convert the movies to a set of individual frames. (word map by tagxedo.com)

12.1.3   Images from a Movie File

In Sect. 12.1.1 we loaded an image sequence into memory where each image came from a separate image file. More commonly image sequences are stored in a movie file format such as MPEG4 or AVI and it may not be practical or possible to keep the whole sequence in memory.

The Toolbox supports reading frames from a movie file stored in any of the popular formats such as AVI, MPEG and MPEG4. For example we can open a movie file

```
>> cam = Movie('traffic_sequence.mpg');
traffic_sequence.mpg
720 x 576 @ 30 fps; 351 frames, 11.7 sec
cur frame 1/351 (skip=1)
```

which returns a `Movie` object that is an instance of a subclass of the `ImageSource` class and therefore polymorphic with the `VideoCamera` class just described. This movie has 350 frames and was captured at 30 frames per second.

The size of each frame within the movie is

```
>> cam.size()
ans =
   720    576
```

and the next frame is read from the movie file by

```
>> im = cam.grab();
>> about(im)
im [uint8] : 576x720x3 (1244160 bytes)
```

which is a 720×576 color image. With these few primitives we can write a very simple movie player

```
1     while 1
2         im = cam.grab;
3         if isempty(im) break; end
4         image(im); drawnow
5     end
```

where the test at line 3 is to detect the end of file, in which case `grab` returns an empty matrix.

The methods `nframes` and `framerate` provide the total number of frames and the number of frames per second. The methods `skiptotime` and `skiptoframe` provide an ability to select particular frames within the movie.

12.1.4   Images from the Web

The term web camera has come to mean *any* USB or Firewire connected local camera but here we use it to refer to an *internet* connected camera that runs a web server that can deliver images on request. There are tens of thousands of these web cameras around the world that are pointed at scenes from the mundane to the spectacular. Given the URL of a webcam from Axis Communications▸ we can acquire an image from a camera anywhere in the world and place it in a matrix in our MATLAB workspace.

For example we can connect to a camera at Dartmouth College in New Hampshire

```
>> cam = AxisWebCamera('http://wc2.dartmouth.edu');
```

which returns an `AxisWebCamera` object which is an instance of a subclass of the `ImageSource` class and therefore polymorphic with the `VideoCamera` and `Movie` classes previously described.

The image size in this case is

```
>> cam.size()
ans =
    480    640
```

and the next image is obtained by

```
>> im = cam.grab();
```

which returns a color image such as the one shown in Fig. 12.3. Webcams are configured by their owner to take pictures periodically, anything from once per second to once per minute. Repeated access will return the same image until the camera takes its next picture.
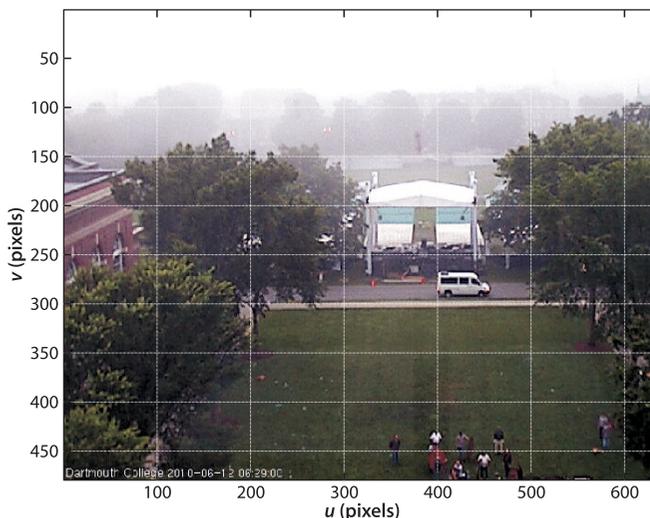
Webcams support a variety of options that can be embedded in the URL and there is no standard for these. This Toolbox function only supports webcams from Axis Communications.

Fig. 12.3.
An image from the Dartmouth University webcam which looks out over the main college green

Aspect ratio is the ratio of an image's width to its height. It varies widely across different imaging and display technologies. For 35 mm film it is $3:2$ (1.5) which matches a $4 \times 6''$ (1.5) print. Other print sizes have different aspect ratios: $5 \times 7''$ (1.4), and $8 \times 10''$ (1.25) which require cropping the vertical edges of the image in order to fit.

TV and early computer monitors used $4:3$ (1.33), for example the ubiquitous 640×480 VGA format. HDTV has settled on $16:9$ (1.78). Modern digital SLR cameras typically use 1.81 which is close to the ratio for HDTV. In movie theatres very-wide images are preferred with aspect ratios of 1.85 or even 2.39. CinemaScope was developed by 20th Century Fox from the work of Henri Chrétien in the 1920s. An anamorphic lens on the camera compresses a wide image into a standard aspect ratio in the camera, and the process is reversed at the projector.

12.1.5   Images from Maps

You can access satellite views and road maps of anywhere on the planet from inside MATLAB. First create an instance of an `EarthView` object

```
>> ev = EarthView('key', YOUR_KEY);
```

where `YOUR_KEY` is your 86-character Google API key◄. To grab a satellite image of my university is simply

```
>> ev.grab(-27.475722,153.0285, 17);
```

which is shown in Fig. 12.4a, and the arguments are latitude, longitude and a zoom level◄. With no output arguments the result will be displayed using `idisp`. If the co-ordinates are unknown we can perform a lookup by name

```
>> ev.grab('QUT brisbane', 17)
```

Instead of a satellite view we can select a road map view

```
>> ev.grab(-27.475722,153.0285, 15, 'map');
```

which shows rich mapping information such as road and place names. A simpler representation is given by

```
>> ev.grab(-27.475722,153.0285, 15, 'roads');
```

which is shown in Fig. 12.4 as a binary image where white pixels correspond to roads and everything else is black – we could use this occupancy grid for robot path planning as discussed in Chap. 5.

12.1.6   Images from Code

When debugging an algorithm it can be very helpful to start with a perfect and simple image before moving on to more challenging real-world images. You could draw such an image with your favorite drawing package and import it to MATLAB, or draw it directly in MATLAB. The Toolbox function `testpattern` generates simple images with a variety of patterns including lines, grids of dots or squares, intensity ramps and intensity sinusoids. For example

```
>> im = testpattern('rampx', 256, 2);
>> im = testpattern('siny', 256, 2);
>> im = testpattern('squares', 256, 50, 25);
>> im = testpattern('dots', 256, 256, 100);
```
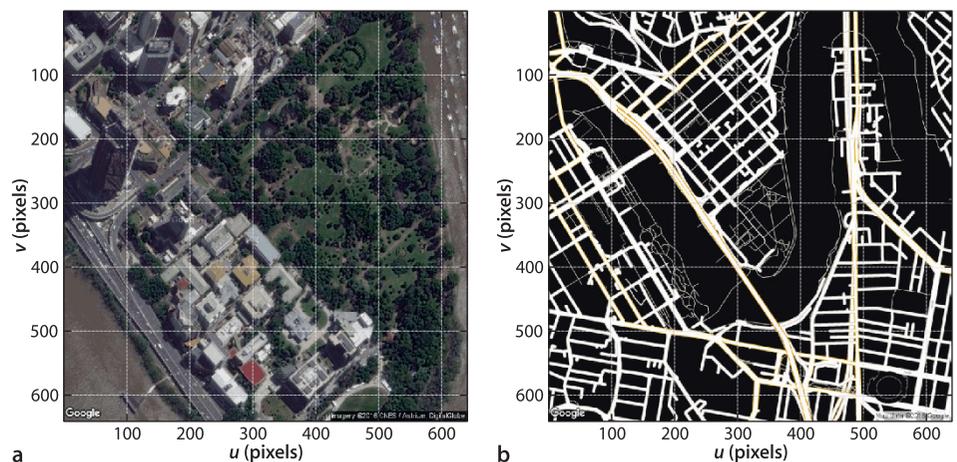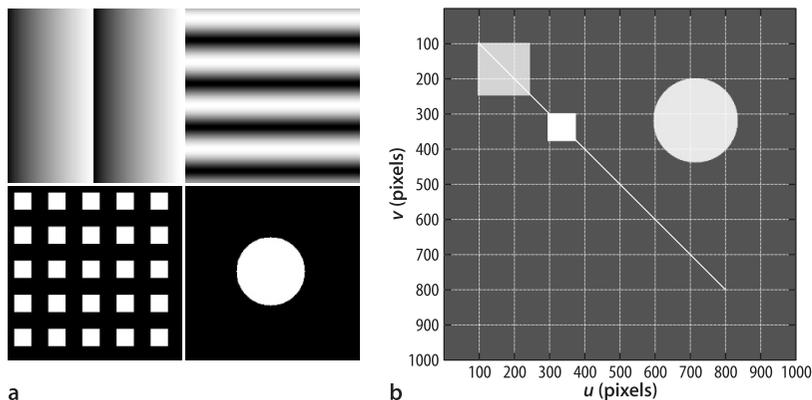
Fig. 12.4.
Aerial views of Brisbane, Australia. **a** Color aerial image; **b** binary image or occupancy grid where white pixels are driveable roads (images provided by Google, CNES/Astrium, Sinclair Knight Merz & Fugro)

Fig. 12.5.
Images from code. **a** Some Tool-box generated test patterns; **b** Simple image created from graphical primitives

are shown in Fig. 12.5a. The second argument is the size of the of the created image, in this case they are all 256×256 pixels, and the remaining arguments are specific to the type of pattern requested. See the online documentation for details.

We can also construct an image from simple graphical primitives.▶ First we create a blank *canvas* containing all black pixels (pixel value of zero)

```
>> canvas = zeros(1000, 1000);
```

An image/matrix can be edited using the command `openvar('canvas')` which brings up a spreadsheet-like interface.

and then we create two squares

```
>> sq1 = 0.5 * ones(150, 150);
>> sq2 = 0.9 * ones(80, 80);
```

The first has pixel values of 0.5 (medium grey) and is 40×40. The second is smaller (just 20×20) but brighter with pixel values of 0.9. Now we can paste these onto the canvas

```
>> canvas = ipaste(canvas, sq1, [100 100]);
>> canvas = ipaste(canvas, sq2, [300 300]);
```

where the last argument specifies the canvas coordinate (u, v) where the pattern will be pasted – the top-left corner of the pattern on the canvas. We can also create a circle

```
>> circle = 0.6 * kcircle(120);
```

of radius 30 pixels with a grey value of 0.6. The Toolbox function `kcircle` returns a square matrix

```
>> size(circle)
ans =
    61    61
```

of zeros with a centered maximal disk of values set to one. We can paste that on to the canvas as well

```
>> canvas = ipaste(canvas, circle, [600, 200]);
```

Finally, we draw a line segment onto our canvas

```
>> canvas = iline(canvas, [100 100], [800 800], 0.8);
```

which extends from (100, 100) to (800, 800) and its pixels are all set to 0.8. The result

```
>> idisp(canvas)
```

is shown in Fig. 12.5b. We can clearly see that the shapes have different brightness, and we note that the line and the circle show the effects of quantization which results in a *steppy* or jagged shape.▶

In computer graphics it is common to apply anti-aliasing where edge pixels and edge-adjacent pixels are set to fractional grey values which give the impression of a smoother line.

> Note that all these functions take coordinates expressed in (u, v) notation not MATLAB row column notation. The top-left pixel is (1, 1) not (0, 0).

12.2 Image Histograms

The distribution of pixel values provides useful information about the quality of the image and the composition of the scene. We obtain the distribution by computing the histogram of the image which indicates the number of times each pixel value occurs. For example the histogram of the image, shown in Fig. 12.8a, is computed and displayed by

```
>> church = iread('church.png', 'grey');
>> ihist( church )
```

and the result is shown in Fig. 12.6a. We see that the grey values (horizontal axis) span the range from 5 to 238 which is close to the full range of possible values. If the image was under-exposed the histogram area would be shifted to the left. If the image was over-exposed the histogram would be shifted to the right and many pixels would have the maximum value. A cumulative histogram is shown in Fig. 12.6b and its use will be discussed in the next section. Histograms can also be computed for color images, in which case the result is three histograms – one for each color channel.

In this case distribution of pixel values is far from uniform and we see that there are three significant peaks. However if we look more closely we see lots of very minor peaks. The concept of a peak depends on the scale at which we consider the data. We can obtain the histogram as a pair of vectors

```
>> [n,v] = ihist(church);
```

where the elements of n are the number of times pixels occur with the value of the corresponding element of v. The Toolbox function peak will automatically find the position of the peaks

```
>> [~,x] = peak(n, v)
>> about x
x [double] : 1x58 (464 bytes)
```

and in this case has found 58 peaks most of which are quite minor. Peaks that are *significant* are not only greater than their immediate neighbors they are greater than all other values *nearby* – the problem now is to specify what we mean by nearby. For example the peaks that are greater than all other values within ±25 pixel values in the horizontal direction are

```
>> [~,x] = peak(n, v, ,scale', 25)
x =
    213    147    41
```

which are the three significant peaks that we observe by eye. The critical part of finding the peaks is choosing the appropriate scale. Peak finding is a topic that we will encounter again later and is also discussed in Appendix J.

Fig. 12.6. Church scene. **a** Histogram, **b** cumulative histogram before and after normalization
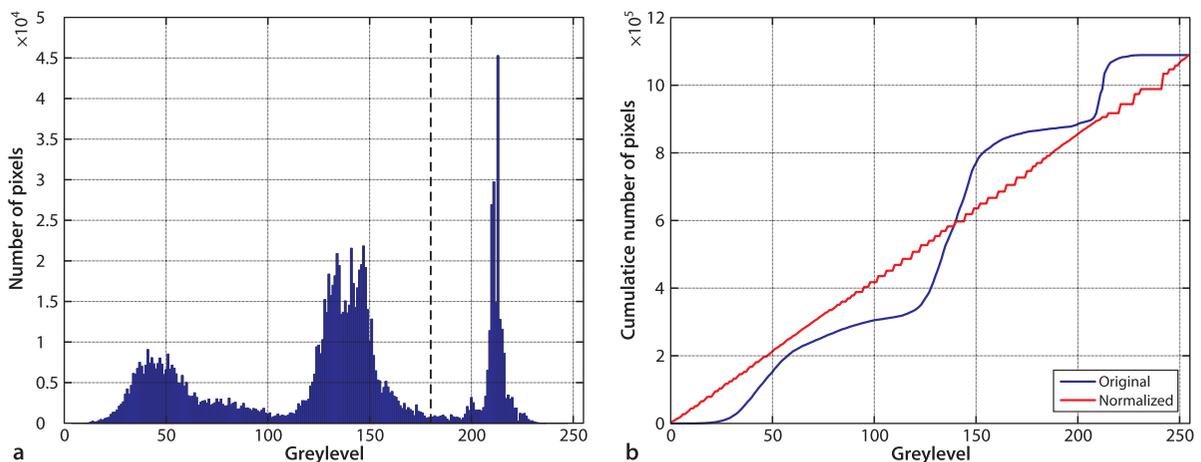
The peaks in the histogram correspond to particular populations of pixels in the image. The lowest peak corresponds to the dark pixels which generally belong to the ground and the roof. The middle peak generally corresponds to the sky pixels, and the highest peak generally corresponds to the white walls. However each of the scene elements has a distribution of grey values and for most real scenes we cannot simply map grey level to a scene element. For example some sky pixels are brighter than some wall pixels, and a very small number of ground pixels are brighter than some sky and wall pixels.

12.3   Monadic Operations

Monadic image-processing operations are shown schematically in Fig. 12.7. The result is an image of the same size $W \times H$ as the input image, and each output pixel is a function of the corresponding input pixel

$$O[u, v] = f(I[u, v]), \quad \forall (u, v) \in I$$

One useful class of monadic functions changes the type of the pixel data. For example to change from `uint8` (integer pixels in the range [0, 255]) to double precision values in the range [0, 1] we use the Toolbox function `idouble`

```
>> imd = idouble(church);
```

and vice versa

```
>> im = iint(imd);
```

A color image has 3-dimensions which we can also consider as a 2-dimensional image where each pixel value is a 3-vector. A monadic operation can convert a color image to a greyscale image where each output pixel value is a scalar representing the luminance of the corresponding input pixel

```
>> grey = imono(flowers);
```

The inverse operation is

```
>> color = icolor(grey);
```

which returns a 3-dimensional color image where each color plane is equal to `grey` – when displayed it still appears as a monochrome image. We can create a color image where the red plane is equal to the input image by

```
>> color = icolor(grey, [1 0 0]);
```

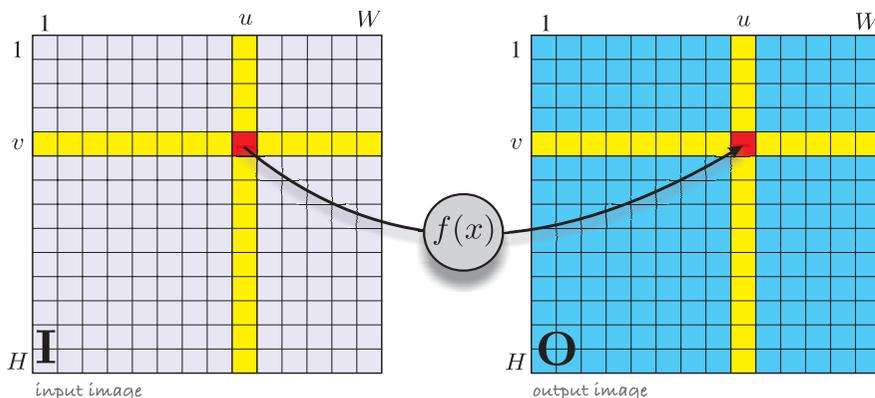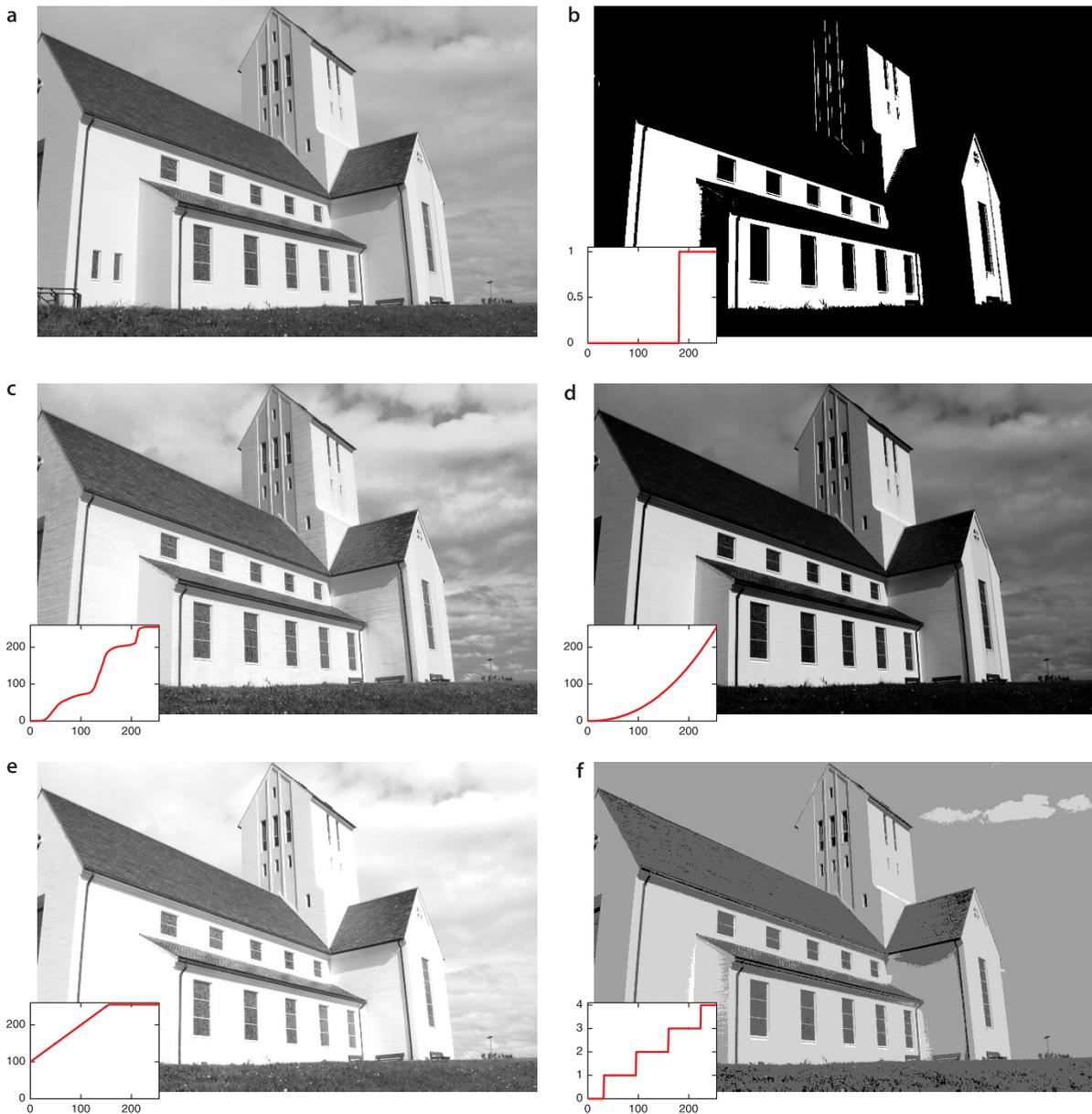which is a red tinted version of the original image.

Fig. 12.7.
Monadic image processing operations. Each output pixel is a function of the corresponding input pixel (shown in *red*)

Fig. 12.8. Some monadic image operations: **a** original, **b** thresholding, **c** histogram normalized, **d** gamma correction, **e** brightness increase, **f** posterization. *Inset* in each figure is a graph showing the mapping from image grey level on the horizontal axis to the output value on the vertical axis

A very common monadic operation is thresholding. This is a logical monadic operation which separates the pixels into two classes according to their intensity

```
>> bright = (church >= 180);
>> idisp(bright)
```

and the resulting image is shown in Fig. 12.8b where all pixels that lie in the interval [180, 255] are shown as white. Such images, where the pixels have only two values are known as binary images. Looking at the image histogram in Fig. 12.6a we see that the grey value of 180 lies midway between the second and third peak which is a good approximation to the optimal strategy for separating pixels belonging to these two populations.

The variable `bright` is of type logical where the pixels have values of only true or false. MATLAB automatically converts these to one and zero respectively when used in arithmetic operations and the `idisp` function does likewise.

Many monadic operations are concerned with altering the distribution of grey levels within the image. Sometimes an image does not span the full range of available grey levels, for example the image is under- or over-exposed. We can apply a linear mapping to the grey-scale values

```
>> im = istretch(church);
```

which ensures that pixel values span the full range▶ which is either [0, 1] or [0, 255] depending on the class of the image.

A more sophisticated version is histogram normalization or histogram equalization

```
>> im = inormhist(church);
```

which is based on the cumulative distribution

```
>> ihist(church, 'cdf');
```

as shown in Fig. 12.6b. Mapping the original image via the normalized cumulative distribution ensures that the cumulative distribution of the resulting image is linear – all grey values occur an equal number of times. The result is shown in Fig. 12.8c and now details of wall and sky texture which had a very small grey-level variation have been accentuated.

> Operations such as `istretch` and `inormhist` can *enhance* the image from the perspective of a human observer, but it is important to remember that no new information has been added to the image. Subsequent image processing steps will not be improved.

As discussed in Sect. 10.3.6 the output of a camera is generally gamma encoded so that the pixel value is a nonlinear function L^γ of the luminance sensed at the photosite. Such images can be gamma decoded by a nonlinear monadic operation

```
>> im = igamm(church, 1/0.45);
```

that raises each pixel to the specified power as shown in Fig. 12.8d, or

```
>> im = igamm(church, 'sRGB');
```

to decode images with the sRGB standard gamma encoding.▶

Another simple nonlinear monadic operation is posterization or banding. This pop-art effect is achieved by reducing the number of grey levels

```
>> idisp( church/64 )
```

as shown in Fig. 12.8f. Since integer division is used the resulting image has pixels with values in the range [0, 3] and therefore just four different shades of grey. Finally, since an image is represented by a matrix any MATLAB element-wise matrix function or operator is a monadic operator, for example unary negation, scalar multiplication or addition, or functions such `abs` or `sqrt`.

12.4   Diadic Operations

Diadic operations are shown schematically in Fig. 12.9. Two input images result in a single output image, and all three images are of the same size. Each output pixel is a function of the corresponding pixels in the two input images

$$O[u, v] = f\left(I_1[u, v], I_2[u, v]\right), \ \ \forall(u, v) \in I_1$$

Examples of useful diadic operations include binary arithmetic operators such as addition, subtraction, element-wise multiplication, or builtin MATLAB diadic matrix functions such as `max`, `min`, `atan2`.

The histogram of such an image will have gaps. If M is the maximum possible pixel value, and $N < M$ is the maximum value in the image then the stretched image will have at most N unique pixel values, meaning that $M - N$ values cannot occur.

The gamma correction has now been applied twice: once by the `igamm` function and once in the display device. This makes the resulting image appear to have unnaturally high contrast.
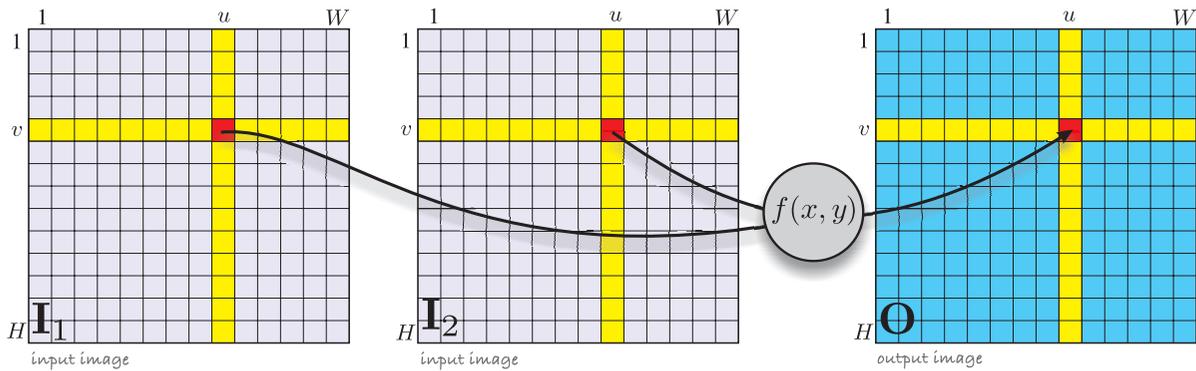
Fig. 12.9. Diadic image processing operations. Each output pixel is a function of the two corresponding input pixels (shown in *red*)

Subtracting one `uint8` image from another results in another `uint8` image even though the result is potentially negative. MATLAB quite properly clamps values to the interval [0, 255] so subtracting a larger number from a smaller number will result in zero not a negative value. With addition a result greater than 255 will be set to 255. To remedy this, the images should be first converted to signed integers using the MATLAB function `cast` or to floating-point values using the Toolbox function `idouble`.

We will illustrate diadic operations with two examples. The first example is chroma-keying – a technique commonly used in television to superimpose the image of a person over some background, for example a weather presenter superimposed over a weather map. The subject is filmed against a blue or green background which makes it quite easy, using just the pixel values, to distinguish between background and the subject. We load an image of a subject taken in front of a green screen

```
>> subject = iread('greenscreen.jpg', 'double');
```

and this is shown in Fig. 12.10a. We compute the chromaticity coordinates Eq. 10.9

```
>> linear = igamm(subject, 'sRGB');
>> [r,g] = tristim2cc(linear);
```

after first converting the gamma encoded color image to linear tristimulus values. In this case g alone is sufficient to distinguish the background pixels. A histogram of values

```
>> ihist(g)
```

shown in Fig. 12.10b indicates a large population of pixels around 0.55 which is the background and another population which belongs to the subject. We can safely say that the subject corresponds to any pixel for which $g < 0.45$ and create a *mask* image

```
>> mask = g < 0.45;
>> idisp(mask)
```

where a pixel is true (equal to one and displayed as white) if it is part of the subject as shown in Fig. 12.10c. We need to apply this mask to all three color planes so we replicate it
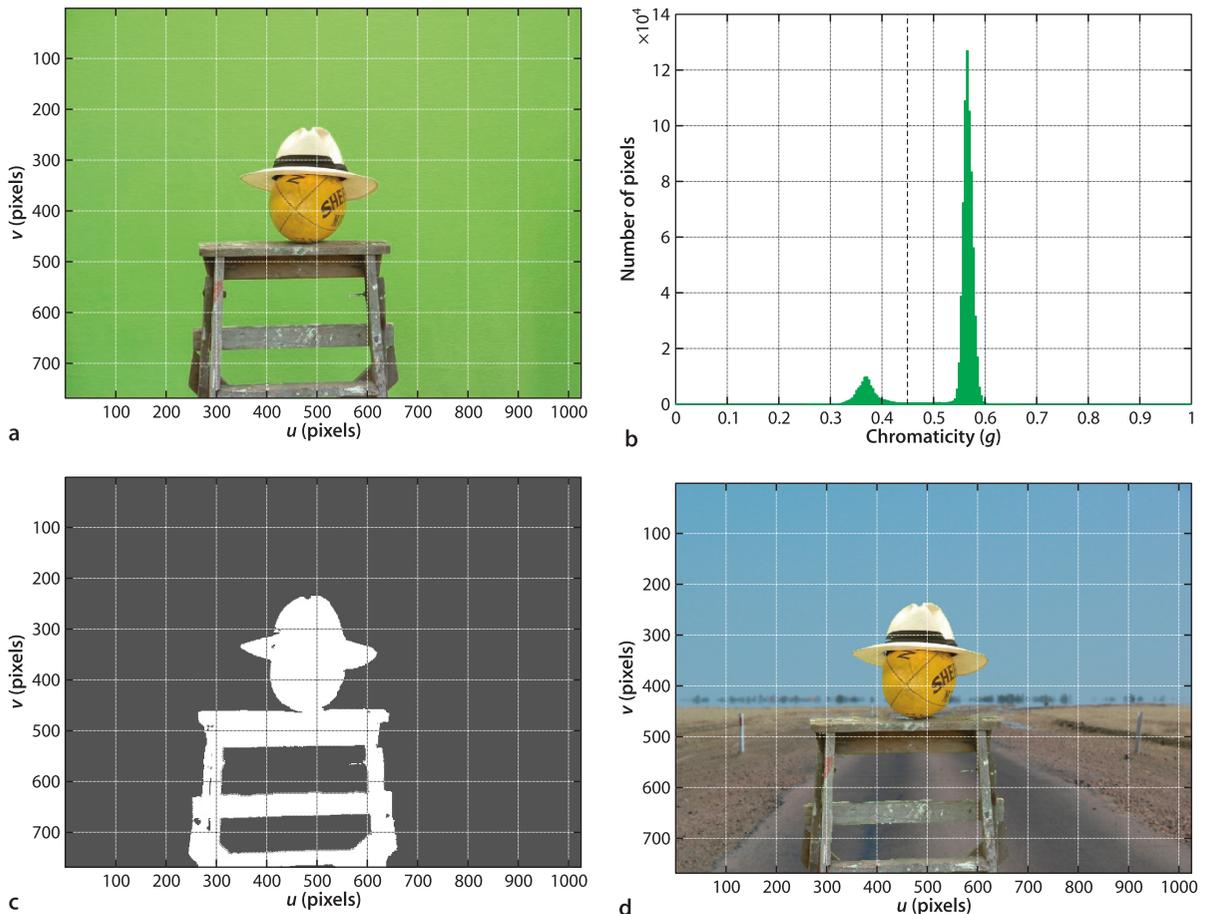
```
>> mask3 = icolor( idouble(mask) );
```

The image of the subject without the background is

```
>> idisp(mask3 .* subject);
```

Next we load the desired background image

```
>> bg = iread('road.png', 'double');
```

and scale and crop it to be the same size as our original image

```
>> bg = isamesize(bg, subject);
```

and display it with a *cutout* for the subject

```
>> idisp(bg .* (1-mask3))
```

Finally we add the subject with no background, to the background with no subject to obtain the subject superimposed over the background

```
>> idisp( subject.*mask3  + bg.*(1-mask3) );
```

which is shown in Fig. 12.10d. The technique will of course fail if the subject contains any colors that match the color of the background.▶ This example could be solved more compactly using the Toolbox per-pixel switching function `ipixswitch`

```
>> ipixswitch(mask, subject, bg);
```

where all arguments are images of the same width and height, and each output pixel is selected from the corresponding pixel in the second or third image according to the logical value of the corresponding pixel in the first image.

Distinguishing foreground objects from the background is an important problem in robot vision but the terms foreground and background are ill-defined and application specific. In robotics we rarely have the luxury of a special background as we did for the chroma-key example. We could instead take a picture of the scene without a foreground object present and consider this to be the background, but that requires that we have special knowledge about when the foreground object is not present. It also assumes that

Fig. 12.10. Chroma-keying. **a** The subject against a green background; **b** a histogram of green chromaticity values; **c** the computed mask image where true is white; **d** the subject *masked* into a background scene (photo courtesy of Fiona Corke)

In the early days of television a blue screen was used. Today a green background is more popular because of problems that occur with blue eyes and blue denim clothing.
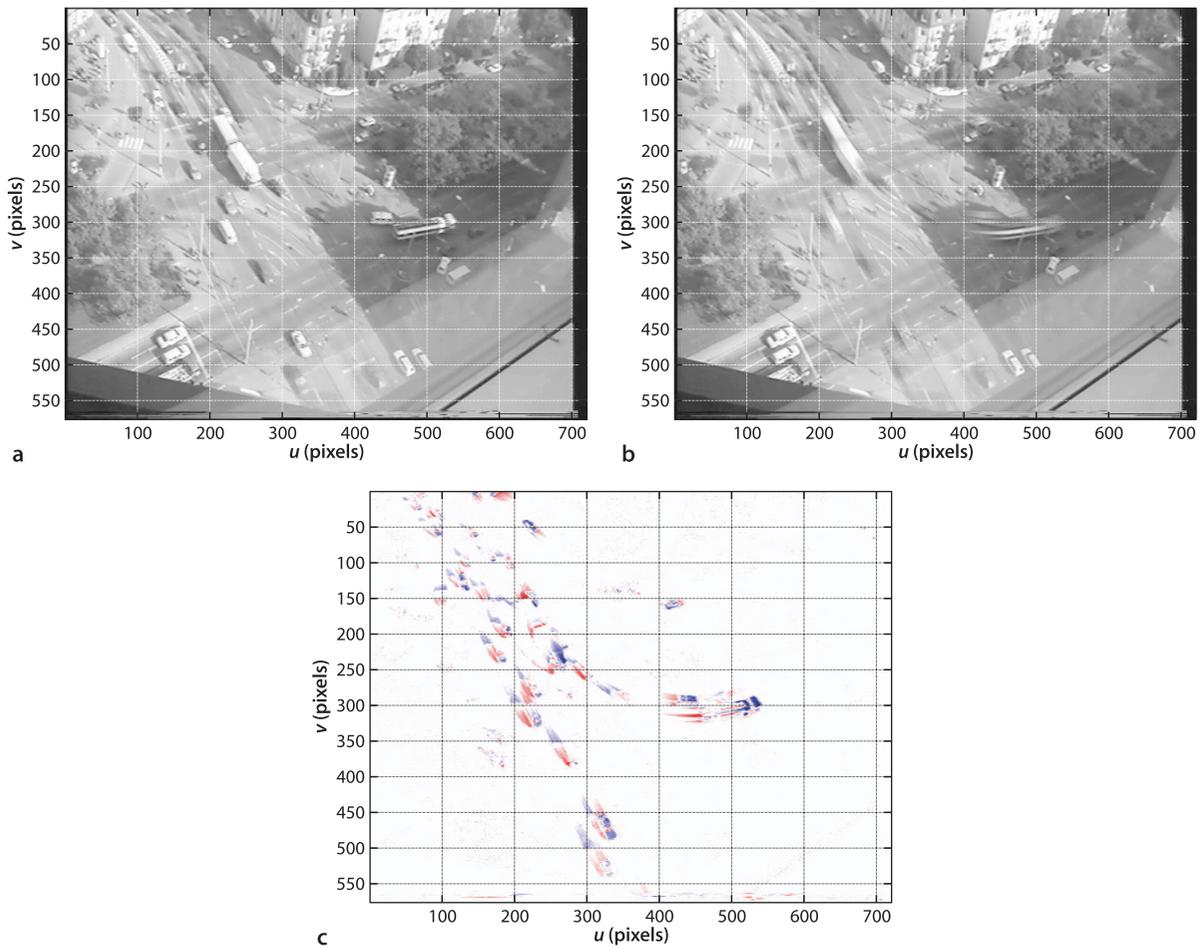
a

b

c

Fig. 12.11. Example of motion detection for the traffic sequence at frame 200. **a** The current image; **b** the estimated background image; **c** the difference between the current and estimated background images where *white* is zero, *red* and *blue* are negative and positive values respectively and magnitude is indicated by *color intensity*

the background does not vary over time. Variation is a significant problem in real-world scenes where ambient illumination and shadows change over quite short time intervals, and the scene may be structurally modified over very long time intervals.

In the next example we process an image sequence and *estimate* the background even though there are a number of objects moving in the scene. We will use a recursive algorithm that updates the estimated background image $\hat{B}$ at each time step based on the previous estimate and the current image

$$\hat{B}\langle k+1\rangle \leftarrow \hat{B}\langle k\rangle + c\left(I\langle k\rangle - \hat{B}\langle k\rangle\right)$$

where k is the time step and $c(\cdot)$ is a monadic image saturation function

$$c(x) = \begin{cases} \sigma, & x > \sigma \\ x, & -\sigma \le x \le \sigma \\ -\sigma, & x < -\sigma \end{cases}$$

To demonstrate this we open a movie showing traffic moving through an intersection

```
>> vid = Movie('traffic_sequence.mpg', 'grey', 'double');
vid =
traffic_sequence.mpg
720 x 576 @ 30 fps; 351 frames, 11.7 sec
cur frame 1/351 (skip=1)
```

and initialize the background to the first image in the sequence

```
>> bg = vid.grab();
```

then the main loop is

```
1   sigma = 0.02;
2   while 1
3      im = vid.grab;
4      if isempty(im) break; end; % end of file?
5      d = im-bg;
6      d = max(min(d, sigma), -sigma);  % apply c(.)
7      bg = bg + d;
8      idisp(bg);
9   end
```

One frame from this sequence is shown in Fig. 12.11a. The estimated background image shown in Fig. 12.11b reveals the static elements of the scene and the moving vehicles have become a faint blur. Subtracting the scene from the estimated background creates an image where pixels are bright where they are different to the background as shown in Fig. 12.11c. Applying a threshold to the absolute value of this difference image shows the area of the image where there is motion. Of course where the cars are stationary for long enough they will become part of the background.

12.5   Spatial Operations

Spatial operations are shown schematically in Fig. 12.12. Each pixel in the output image is a function of all pixels in a *region* surrounding the corresponding pixel in the input image

$$O[u,v] = f\big(I[u+i, v+j]\big), \quad \forall (i,j) \in \mathcal{W}, \quad \forall (u,v) \in I$$

where $\mathcal{W}$ is known as the window, typically a $w \times w$ square region with odd side length $w = 2h + 1$ where $h \in \mathbb{Z}^+$ is the half-width. In Fig. 12.12 the window includes all pixels in the red shaded region. Spatial operations are powerful because of the variety of possible functions $f(\cdot)$, linear or nonlinear, that can be applied. The remainder of this section discusses linear spatial operators such as smoothing and edge detection, and some nonlinear functions such as rank filtering and template matching. The following section covers a large and important class of nonlinear spatial operators known as mathematical morphology.

12.5.1   Linear Spatial Filtering

A very important linear spatial operator is correlation

$$\mathbf{O}[u,v] = \sum_{(i,j)\in\mathcal{W}} I[u+i, v+j]K[i,j], \; \forall (u,v) \in \mathbf{I} \tag{12.1}$$

where $K \in \mathbb{R}^{w \times w}$ is the kernel and the elements are referred to as the filter coefficients. For every output pixel the corresponding window of pixels from the input image $\mathcal{W}$ is multiplied element-wise with the kernel K. The center of the window and kernel is considered to be coordinate $(0, 0)$ and $i, j \in [-h, h] \subset \mathbb{Z} \times \mathbb{Z}$. This can be considered as the weighted sum of pixels within the window where the weights are defined by the kernel K. Correlation is often written in operator form as

$$\mathbf{O} = K \otimes \mathbf{I}$$

A closely related operation is convolution

$$\mathbf{O}[u,v] = \sum_{(i,j)\in\mathcal{W}} I[u-i, v-j]K[i,j], \quad \forall (u,v) \in \mathbf{I} \tag{12.2}$$
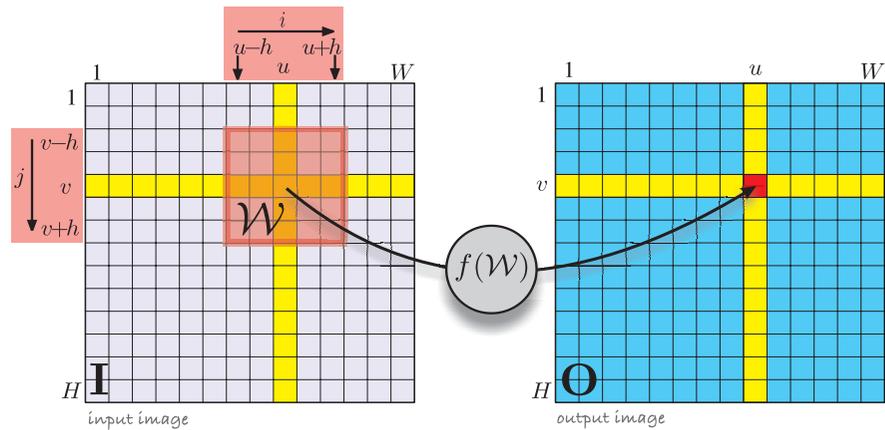
Fig. 12.12.
Spatial image processing operations. The *red shaded region* shows the window $\mathcal{W}$ that is the set of pixels used to compute the output pixel (show in *red*)

where $K \in \mathbb{R}^{w \times w}$ is the convolution kernel. Note that the sign of the i and j indices has changed in the first term. Convolution is often written in operator form as

$$\mathbf{O} = K * \mathbf{I}$$

As we will see convolution is the workhorse of image processing and the kernel K can be chosen to perform functions such as smoothing, gradient calculation or edge detection.

Convolution is computationally expensive – an $N \times N$ input image with a $w \times w$ kernel requires $w^2 N^2$ multiplications and additions. In the Toolbox convolution is performed using the function `iconvolve`

```
O = iconvolve(K, I);
```

If `I` has multiple color planes then so will the output image – each output color plane is the convolution of the corresponding input plane with the kernel `K`.

12.5.1.1 Smoothing

Consider a convolution kernel which is a square 21×21 matrix containing equal elements

```
>> K = ones(21,21) / 21^2;
```

and of unit volume, that is, its values sum to one. The result of convolving an image with this kernel is an image where each output pixel is the mean of the pixels in

Correlation or convolution. These two terms are often used loosely and they have similar, albeit distinct, definitions. Convolution is the spatial domain equivalent of frequency domain multiplication and the kernel is the impulse response of a frequency domain filter. Convolution also has many useful mathematical properties outlined in the adjacent box. The difference in indexing between Eq. 12.1 and Eq. 12.2 is equivalent to reflecting the kernel – flipping it horizontally and vertically about its center point. Many kernels are symmetric in which case correlation and convolution yield the same result. However edge detection is always based on nonsymmetric kernels so we must take care to apply convolution. We will only use correlation for template matching in Sect. 12.5.2.

Properties of convolution. Convolution obeys the familiar rules of algebra, it is commutative

$$A * B = B * A$$

associative

$$A * B * C = (A * B) * C = A * (B * C)$$

distributive (superposition applies)

$$A * (B + C) = A * B + A * C$$

linear

$$A * (\alpha B) = \alpha (A * B)$$

and shift invariant – the spatial equivalent of time invariance in 1D signal processing – the result of the operation is the same everywhere in the image.
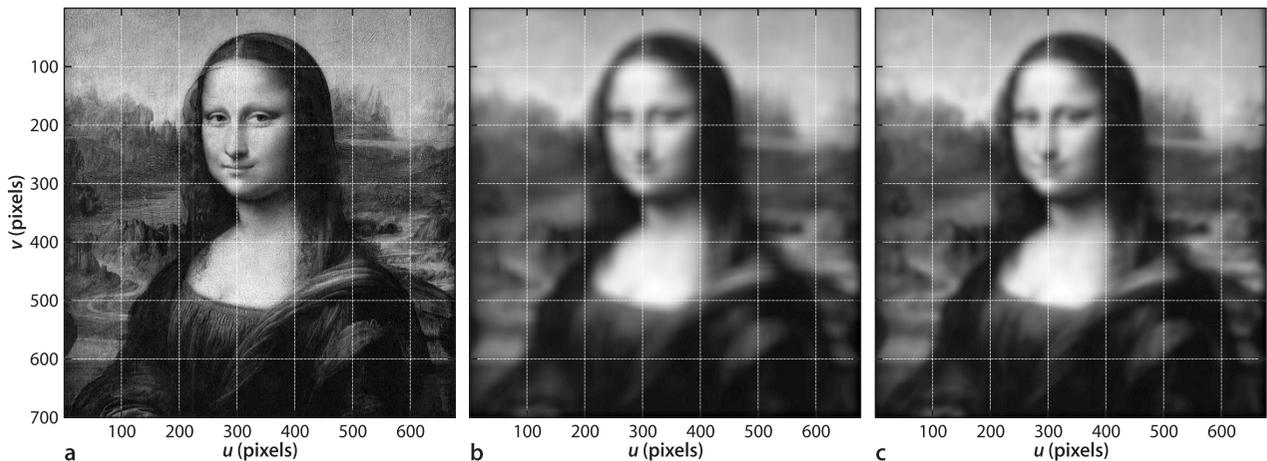
a   *u* (pixels)   b   *u* (pixels)   c   *u* (pixels)

Fig. 12.13. Smoothing. **a** Original image; **b** smoothed with a 21×21 averaging kernel; **c** smoothed with a 31×31 Gaussian $G(\sigma = 5)$ kernel

a corresponding 21×21 neighborhood in the input image. As you might expect this averaging

```
>> mona = iread('monalisa.png', 'double', 'grey');
>> idisp( iconvolve(mona, K) );
```

leads to smoothing, blurring or *defocus*▶ which we see in Fig. 12.13b. Looking very carefully we will see some faint horizontal and vertical lines – an artifact known as ringing. A more suitable kernel for smoothing is the 2-dimensional Gaussian function

Defocus involves a kernel which is a 2-dimensional Airy pattern or sinc function. The Gaussian function is similar in shape, but is always positive whereas the Airy pattern has low amplitude negative going rings.
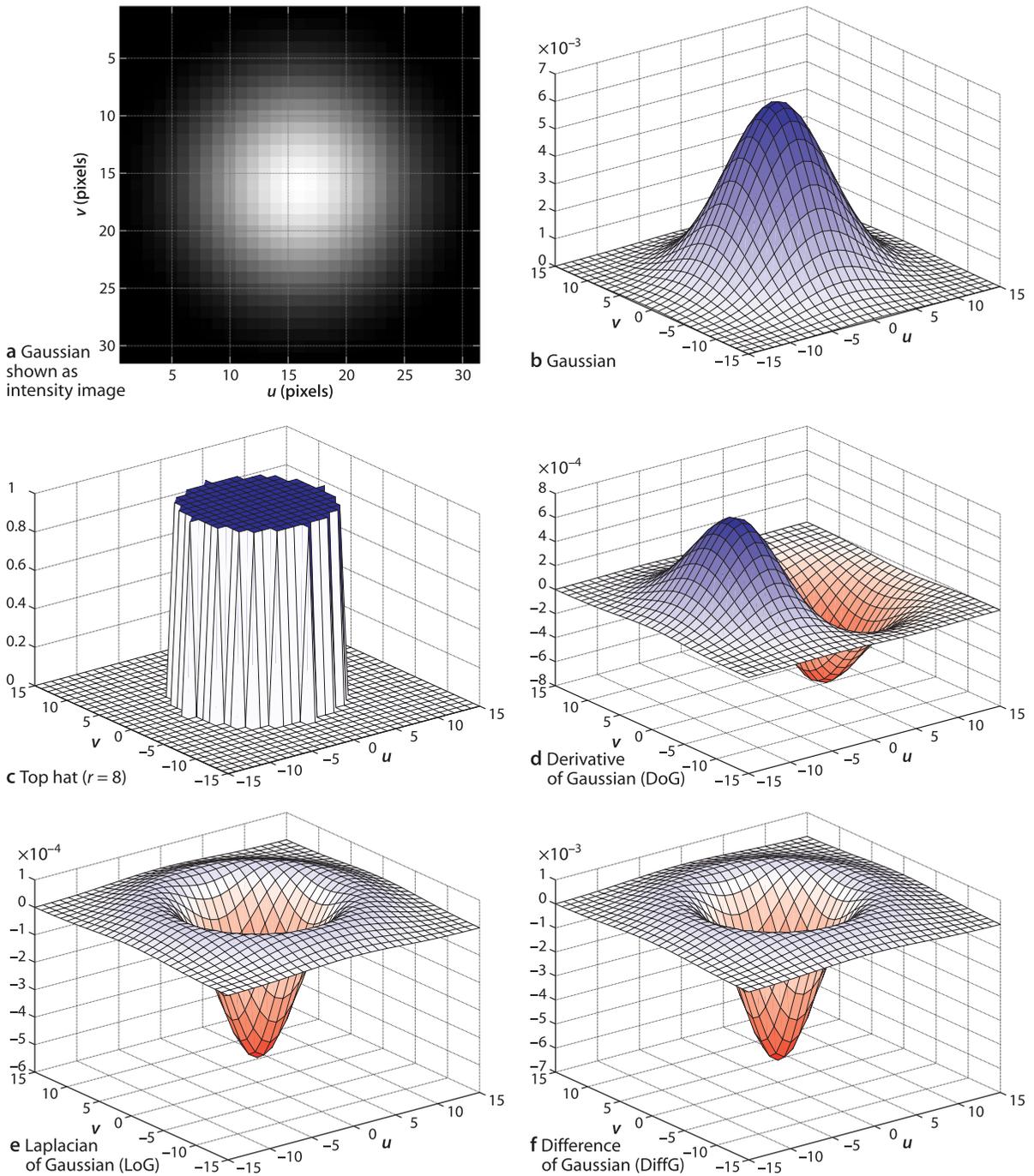
$$\mathbf{G}(u, v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2 + v^2}{2\sigma^2}} \tag{12.3}$$

which is symmetric about the origin and the volume under the curve is unity. The spread of the Gaussian is controlled by the standard deviation parameter σ. Applying this kernel to the image

```
>> K = kgauss(5);
>> idisp( iconvolve(mona, K) );
```

produces the result shown in Fig. 12.13c. Here we have specified the standard deviation of the Gaussian to be 5 pixels. The discrete approximation to the Gaussian is

```
>> about(K)
K [double] : 31x31 (7688 bytes)
```

a 31×31 kernel. Smoothing can be achieved conveniently using the Toolbox function `ismooth`

```
>> idisp( ismooth(mona, 5) )
```

Blurring is a counter-intuitive image processing operation since we typically go to a lot of effort to obtain a clear and crisp image. To deliberately *ruin it* seems, at face value, somewhat reckless. However as we will see later, Gaussian smoothing turns out to be extremely useful.

The kernel is itself a matrix and therefore we can display it as an image

```
>> idisp( K );
```

which is shown in Fig. 12.14a. We clearly see the large value at the center of the kernel and that it falls off smoothly in all directions. We can also display the kernel as a surface

```
>> surfl(-15:15, -15:15, K);
```

a Gaussian shown as intensity image

b Gaussian

c Top hat ($r = 8$)

d Derivative of Gaussian (DoG)

e Laplacian of Gaussian (LoG)

f Difference of Gaussian (DiffG)

Fig. 12.14. Gallery of commonly used convolution kernels. $h = 15$, $\sigma = 5$

as shown in Fig. 12.14b. A crude approximation to the Gaussian is the top hat kernel which is cylinder with vertical sides rather than a smooth and gentle fall off in amplitude. The function `kcircle` creates a kernel which can be considered a unit height cylinder of specified radius

```
>> K = kcircle(8, 15);
```

as shown in Fig. 12.14c. The arguments specify a radius of 8 pixels within a window of half width $h = 15$.

How wide is my Gaussian? When choosing a Gaussian kernel we need to consider the standard deviation, usually defined by the task, and the dimensions of the kernel $\mathcal{W} \in \mathbb{R}^{w \times w}$ that contains the discrete Gaussian function. Computation time is proportional to w^2 so ideally we want the window to be no bigger than it needs to be. The Gaussian decreases monotonically in all directions but never reaches zero. Therefore we choose the half-width h of the window such that value of the Gaussian is less than some threshold outside the $w \times w$ convolution window.

At the edge of the window, a distance h from the center, the value of the Gaussian will be $e^{-h^2/2\sigma^2}$. For $\sigma = 1$ and $h = 2$ the Gaussian will be $e^{-2} \approx 0.14$, for $h = 3$ it will be $e^{-4.5} \approx 0.01$, and for $h = 4$ it will be $e^{-8} \approx 3.4 \times 10^{-4}$. If h is not specified the Toolbox chooses $h = 3\sigma$. For $\sigma = 1$ that is a 7×7 window which contains all values of the Gaussian greater than 1% of the peak value.

Properties of the Gaussian. The Gaussian function $G(\cdot)$ has some special properties. The convolution of two Gaussians is another Gaussian

$$\mathbf{G}(\sigma_1) * \mathbf{G}(\sigma_2) = \mathbf{G}\left(\sqrt{\sigma_1^2 + \sigma_2^2}\right)$$

For the case where $\sigma_1 = \sigma_2 = \sigma$ then

$$\mathbf{G}(\sigma) * \mathbf{G}(\sigma) = \mathbf{G}\left(\sqrt{2}\sigma\right)$$

The 2-dimensional Gaussian is separable – it can be written as the product of two 1-dimensional Gaussians

$$\mathbf{G}(u, v) = \left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{u^2}{2\sigma^2}}\right)\left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{v^2}{2\sigma^2}}\right)$$

This implies that convolution with a 2-dimensional Gaussian can be computed by convolving each row with a 1-dimensional Gaussian, and then each column. The total number of operations is reduced to $2wN^2$, better by a factor of w. A Gaussian also has the same shape in the spatial and frequency domains.

12.5.1.2   Boundary Effects

A difficulty with all spatial operations occurs when the window is close to the edge of the input image as shown in Fig. 12.15. In this case the output pixel is a function of a window that contains pixels *beyond the edge* of the input image – these pixels have no defined value. There are several common remedies to this problem. Firstly, we can assume the pixels beyond the image have a particular value. A common choice is zero and this is the default behavior implemented by the Toolbox function `iconvolve`. We can see the effect of this in Fig. 12.13 where the borders of the smoothed image are dark due to the influence of these zeros.

Another option is to consider that the result is invalid when the window crosses the boundary of the image. Invalid output pixels are shown hatched out in Fig. 12.15. The result is an output image of size $(W - 2h) \times (H - 2h)$ which is slightly smaller than the input image. This option can be selected by passing the option `'valid'` to `iconvolve`.

12.5.1.3   Edge Detection

Frequently we are interested in finding the edges of objects in a scene. Consider the image

```
>> castle = iread('castle.png', 'double', 'grey');
```

shown in Fig. 12.16a. It is informative to look at the pixel values along a 1-dimensional profile through the image. A horizontal profile of the image at $v = 360$ is

```
>> p = castle(360,:);
```

which is a vector that we can plot

```
>> plot(p);
```

against the horizontal coordinate u in Fig. 12.16b. The clearly visible tall spikes correspond to the white letters and other markings on the sign. Looking at one of the spikes more closely, Fig. 12.16c, we see the intensity profile across the vertical stem of the letter T. The background intensity ≈ 0.3 and the bright intensity ≈ 0.9 but will depend on lighting levels. However the very rapid increase over the space of just a few pixels is distinctive and a more reliable indication of an edge than any decision based on the actual grey levels.

The first-order derivative along this cross-section is

$$p'[v] = p[v] - p[v - 1]$$
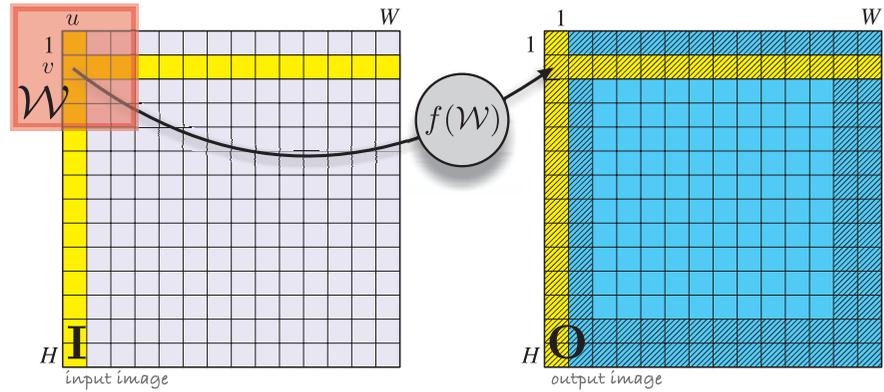
Fig. 12.15.
For the case where the window $\mathcal{W}$ *falls off the edge* of the input image the output pixel at (u, v) is not defined. The *hatched pixels* in the output image are all those for which the output value is not defined
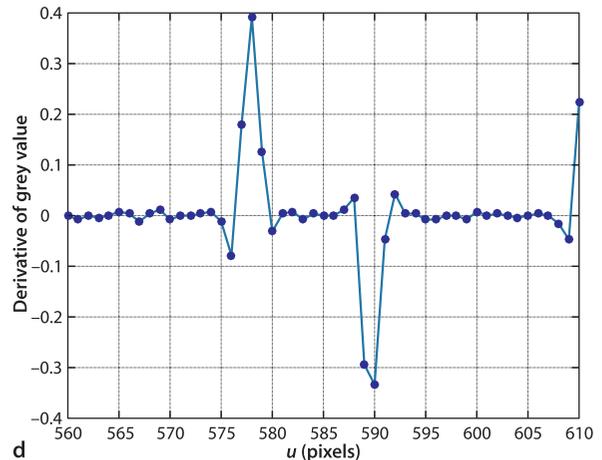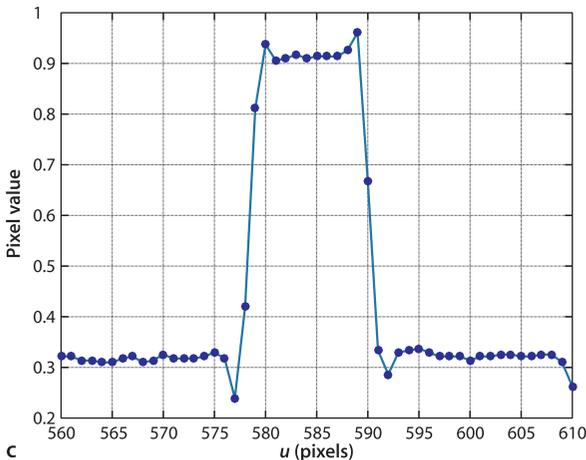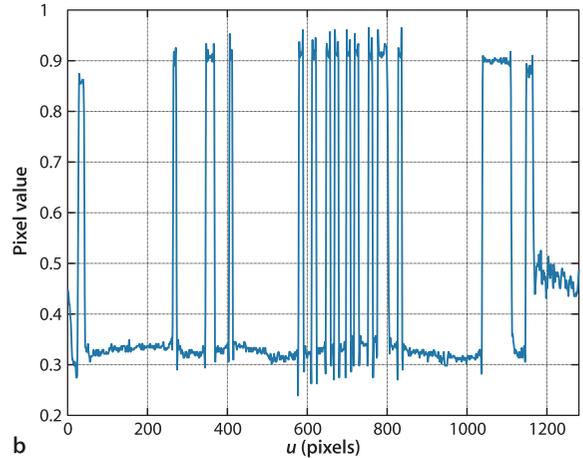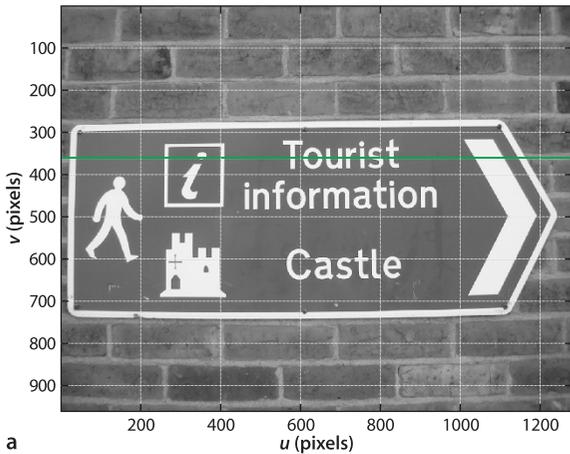
Fig. 12.16. Edge intensity profile.
a Original image; **b** greylevel profile along horizontal line $v = 360$;
c closeup view of the spike at $u \approx 580$; **d** derivative of **c** (image from the ICDAR 2005 OCR dataset; Lucas 2005)

which can be computed using the MATLAB function `diff`

```
>> plot(diff(p))
```

and is shown in Fig. 12.16d. The signal is nominally zero with clear nonzero responses at the edges of an object, in this case the edges of the stem of the letter T.

The derivative at point v can also be written as a *symmetrical* first-order difference

$$p'[v] = \frac{1}{2}\big(p[v+1] - p[v-1]\big)$$
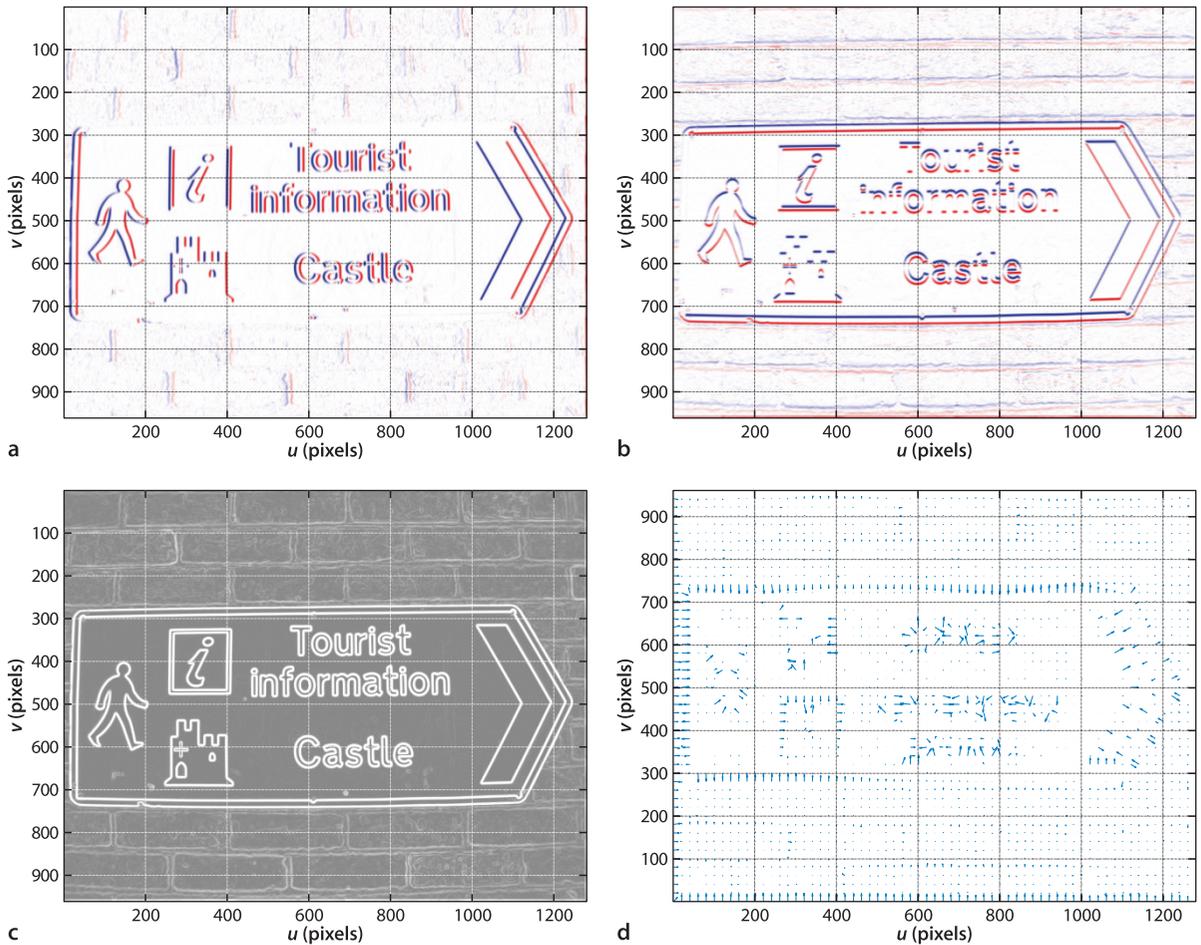
a

b

c

d

Fig. 12.17. Edge gradient. **a** u-direction gradient; **b** v-direction gradient; **c** gradient magnitude; **d** gradient direction. Gradients shown with *blue* as positive, *red* as negative and *white* as zero

which is equivalent to convolution with the 1-dimensional kernel

$$K = \begin{pmatrix} \frac{1}{2} & 0 & -\frac{1}{2} \end{pmatrix}$$

Convolving the image with this kernel

```
>> K = [0.5 0 -0.5];
>> idisp( iconvolve(castle, K), 'invsigned')
```

produces a result very similar to that shown in Fig. 12.17a in which vertical edges, high horizontal gradients, are clearly seen.

> Since this kernel has signed values the result of the convolution will also be signed, that is, the gradient at a pixel can be positive or negative as shown in Fig. 12.17a,b. `idisp` always displays the minimum, most negative, value as black and the maximum, most positive, value as white. Zero would therefore appear as middle grey. The `'signed'` option to `idisp` uses red and blue shading to clearly indicate sign – zero is black, negative pixels are red, positive pixels are blue and the intensity of the color is proportional to pixel magnitude. The `'invsigned'` option is similar except that zero is indicated by white.

Many convolution kernels have been proposed for computing horizontal gradient. A popular choice is the Sobel kernel▶

This kernel is commonly written with the signs reversed which is correct for correlation. For convolution the kernel must be written as shown here.

```
>> Du = ksobel
Du =
    0.1250         0   -0.1250
    0.2500         0   -0.2500
    0.1250         0   -0.1250
```

and we see that each row is a scaled version of the 1-dimensional kernel K defined above. The overall result is a weighted sum of the horizontal gradient for the current row, and the rows above and below. Convolving our image with this kernel

```
>> idisp( iconvolve(castle, Du), 'invsigned')
```

generates the horizontal gradient image shown in Fig. 12.17a which highlights vertical edges. Vertical gradient is computed using the transpose of the kernel

```
>> idisp( iconvolve(castle, Du'), 'invsigned')
```

and highlights horizontal edges◀ as shown in Fig. 12.17b. The notation used for gradients varies considerably in the literature. Most commonly the horizontal and vertical gradient are denoted respectively as $\partial \mathbf{I}/\partial u$, $\partial \mathbf{I}/\partial v$; $\nabla_u \mathbf{I}$, $\nabla_v \mathbf{I}$ or $\mathbf{I}_u$, $\mathbf{I}_v$. In operator form this is written

$$\mathbf{I}_u = \mathbf{D} * \mathbf{I}$$
$$\mathbf{I}_v = \mathbf{D}^T * \mathbf{I}$$

where $\mathbf{D}$ is a derivative kernel such as Sobel.

Taking the derivative of a signal accentuates high-frequency noise, and all images have noise as discussed on page 364. At the pixel level noise is a stationary random process – the values are not correlated between pixels. However the features that we are interested in such as edges have correlated changes in pixel value over a larger spatial scale as shown in Fig. 12.16c. We can reduce the effect of noise by smoothing the image before taking the derivative

$$\mathbf{I}_u = \mathbf{D} * \big(\mathbf{G}(\sigma) * \mathbf{I}\big)$$

Instead of convolving the image with the Gaussian and *then* the derivative, we exploit the associative property of convolution to write

$$\mathbf{I}_u = \mathbf{D} * \big(\mathbf{G}(\sigma) * I\big) = \underbrace{\big(\mathbf{D} * \mathbf{G}(\sigma)\big)}_{\text{DoG}} * \mathbf{I}$$

Filters can be designed to respond to edges at any arbitrary angle. The Sobel kernel itself can be considered as an image and rotated using `irotate`. *To obtain angular precision generally requires a larger kernel is required such as that generated by* `kdgauss`.

Carl Friedrich Gauss (1777–1855) was a German mathematician who made major contributions to fields such as number theory, differential geometry, magnetism, astronomy and optics. He was a child prodigy, born in Brunswick, Germany, the only son of uneducated parents. At the age of three he corrected, in his head, a financial error his father had made, and made his first mathematical discoveries while in his teens. Gauss was a perfectionist and a hard worker but not a prolific writer. He refused to publish anything he did not consider complete and above criticism. It has been suggested that mathematics could have been advanced by fifty years if he had published all of his discoveries. According to legend Gauss was interrupted in the middle of a problem and told that his wife was dying – he responded "Tell her to wait a moment until I am through".

The normal distribution, or Gaussian function, was not one of his achievements. It was first discovered by de Moivre in 1733 and again by Laplace in 1778. The SI unit for magnetic flux density is named in his honor.

We convolve the image with the *derivative of the Gaussian* (DoG) which can be obtained numerically by

```
Gu = iconvolve( Du, kgauss(sigma) , 'full');
```

or analytically by taking the derivative, in the *u*-direction, of the Gaussian Eq. 12.3 yielding

$$\mathbf{G}_u(u,v) = -\frac{u}{2\pi\sigma^4} e^{-\frac{u^2+v^2}{2\sigma^2}} \tag{12.4}$$

which is computed by the Toolbox function `kdgauss` and is shown in Fig. 12.14d.

The standard deviation σ controls the *scale* of the edges that are detected. For large σ, which implies increased smoothing, edges due to fine texture will be attenuated leaving only the edges of large features. This ability to find edges at different spatial scale is important and underpins the concept of scale space that we will discuss in Sect. 13.3.2. Another interpretation of this operator is as a spatial *bandpass filter* since it is a cascade of a low-pass filter (smoothing) with a high-pass filter (differentiation).

Computing the horizontal and vertical components of gradient at each pixel

```
>> Iu = iconvolve( castle, kdgauss(2) );
>> Iv = iconvolve( castle, kdgauss(2)' );
```

allows us to compute the magnitude of the gradient at each pixel

```
>> m = sqrt( Iu.^2 + Iv.^2 );
```

This *edge-strength* image shown in Fig. 12.17c reveals the edges very distinctly. The direction of the gradient at each pixel is

```
>> th = atan2( Iv, Iu);
```

and is best viewed as a sparse quiver plot

```
>> quiver(1:20:numcols(th), 1:20:numrows(th), ...
       Iu(1:20:end,1:20:end), Iv(1:20:end,1:20:end))
```

as shown in Fig. 12.17d. The edge direction plot is much noisier than the magnitude plot. Where the edge gradient is strong, on the border of the sign or the edges of letters, the direction is normal to the edge, but the fine-scale brick texture appears as almost random edge direction. The gradient images can be computed conveniently using the Toolbox function

```
>> [du,dv] = isobel( castle, kdgauss(2) );
```

where the last argument overrides the default Sobel kernel.

A well known and very effective edge detector is the Canny edge operator. It uses the edge magnitude and direction that we have just computed and performs two additional steps. The first is nonlocal maxima suppression. Consider the gradient magnitude image of Fig. 12.17c as a 3-dimensional surface where height is proportional to brightness as shown in Fig. 12.18. We see a series of hills and ridges and we wish to find the

Pierre-Simon Laplace (1749–1827) was a French mathematician and astronomer who consolidated the theories of mathematical astronomy in his five volume *Mécanique Céleste* (Celestial Mechanics). While a teenager his mathematical ability impressed d'Alembert who helped to procure him a professorship. When asked by Napoleon why he hadn't mentioned God in his book on astronomy he is reported to have said "Je n'avais pas besoin de cette hypothèse-là" ("I have no need of that hypothesis"). He became a count of the Empire in 1806 and later a marquis.

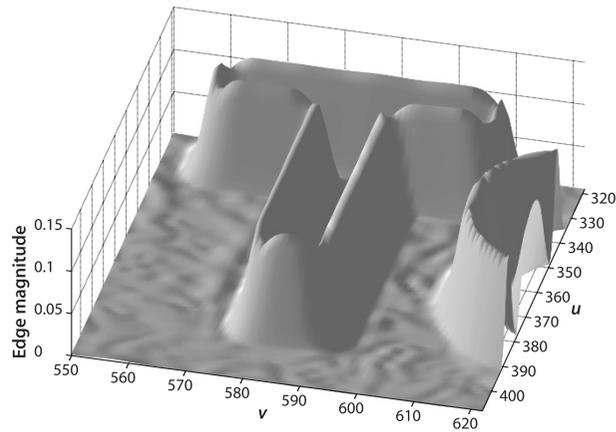The Laplacian operator, a second-order differential operator, and the Laplace transform are named after him.

Fig. 12.18.
Closeup of gradient magnitude around the letter T shown as a 3-dimensional surface
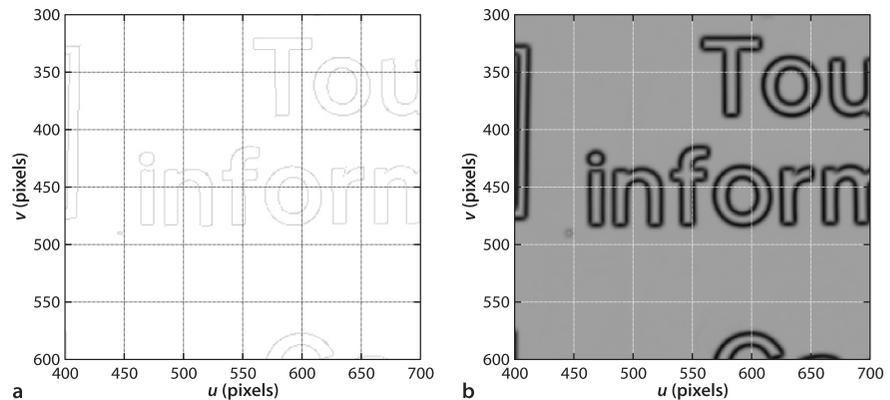

Fig. 12.19.
Comparison of two edge operators: **a** Canny operator with default parameters; **b** Magnitude of derivative of Gaussian kernel ($\sigma = 2$). The |DoG| operator requires less computation than Canny but generates thicker edges. For both cases results are shown *inverted*, *white* is zero

pixels that lie along the *ridge lines*. By examining pixel values in a local neighborhood *normal* to the edge direction, that is in the direction of the edge gradient, we can find the maximum value and set all other pixels to zero. The result is a set of nonzero pixels corresponding to peaks and ridge lines. The second step is hysteresis thresholding. For each nonzero pixel that exceeds the upper threshold a chain is created of adjacent pixels that exceed the lower threshold. Any other pixels are set to zero.

To apply the Canny operator to our example image is straightforward

```
>> edges = icanny(castle, 2);
```

and returns an image where the edges are marked by nonzero intensity values corresponding to gradient magnitude at that pixel as shown in Fig. 12.19a. We observe that the edges are much thinner than those for the magnitude of derivative of Gaussian operator which is shown in Fig. 12.19b. In this example $\sigma = 2$ for the derivative of Gaussian operation. The hysteresis threshold parameters can be set with optional arguments.

Difference of Gaussians. The Laplacian of Gaussian can be approximated by the difference of two Gaussian functions

$$\mathbf{DoG}(u, v; \sigma_1, \sigma_1) = \mathbf{G}(\sigma_1) - \mathbf{G}(\sigma_2) = \frac{1}{2\pi\sigma_1^2\sigma_2^2}\left(\sigma_2^2 e^{-\frac{u^2+v^2}{2\sigma_1^2}} - \sigma_1^2 e^{-\frac{u^2+v^2}{2\sigma_2^2}} \right)$$

where $\sigma_1 > \sigma_2$ and commonly $\sigma_1 = 1.6\sigma_2$. This is computed by the Toolbox function kdog. Figure 12.13e and f shows the LoG and DiffG kernels respectively.

This approximation is useful in scale-space sequences which will be discussed in Sect. 13.3.2. Consider an image sequence $\mathbf{I}\langle k\rangle$ where $\mathbf{I}\langle k+1\rangle = \mathbf{G}(\sigma) \otimes \mathbf{I}\langle k\rangle$, that is, the images are increasingly smoothed. The difference between any two images in the sequence is therefore equivalent to DiffG($\sqrt{2\sigma}, \sigma$) applied to the original image.

So far we have considered an edge as a point of high gradient, and nonlocal maxima suppression has been used to *search* for the maximum value in local neighborhoods. An alternative means to find the point of maximum gradient is to compute the second derivative and determine where this is zero. The Laplacian operator

$$\nabla^2 \mathbf{I} = \frac{\partial^2 \mathbf{I}}{\partial u^2} + \frac{\partial^2 \mathbf{I}}{\partial v^2} = \mathbf{I}_{uu} + \mathbf{I}_{vv} \tag{12.5}$$

is the sum of the second spatial derivative in the horizontal and vertical directions. For a discrete image this can be computed by convolution with the Laplacian kernel

```
>> L = klaplace()
L =
     0     1     0
     1    -4     1
     0     1     0
```

which is isotropic – it responds equally to edges in any direction. The second derivative is even more sensitive to noise than the first derivative and is again commonly used in conjunction with a Gaussian smoothed image

$$\nabla^2 \mathbf{I} = \mathbf{L} * \big(\mathbf{G}(\sigma) * \mathbf{I}\big) = \underbrace{\big(\mathbf{L} * \mathbf{G}(\sigma)\big)}_{\text{LoG}} * \mathbf{I} \tag{12.6}$$

which we combine into the Laplacian of Gaussian kernel (LoG), and **L** is the Laplacian kernel given above. This can be written analytically as

Fig. 12.20. Laplacian of Gaussian. **a** Laplacian of Gaussian; **b** close-up of **a** around the letter T where *blue* and *red* colors indicate positive and negative values respectively; **c** a horizontal cross-section of the LoG through the stem of the T; **d** closeup of the zero-crossing detector output at the letter T
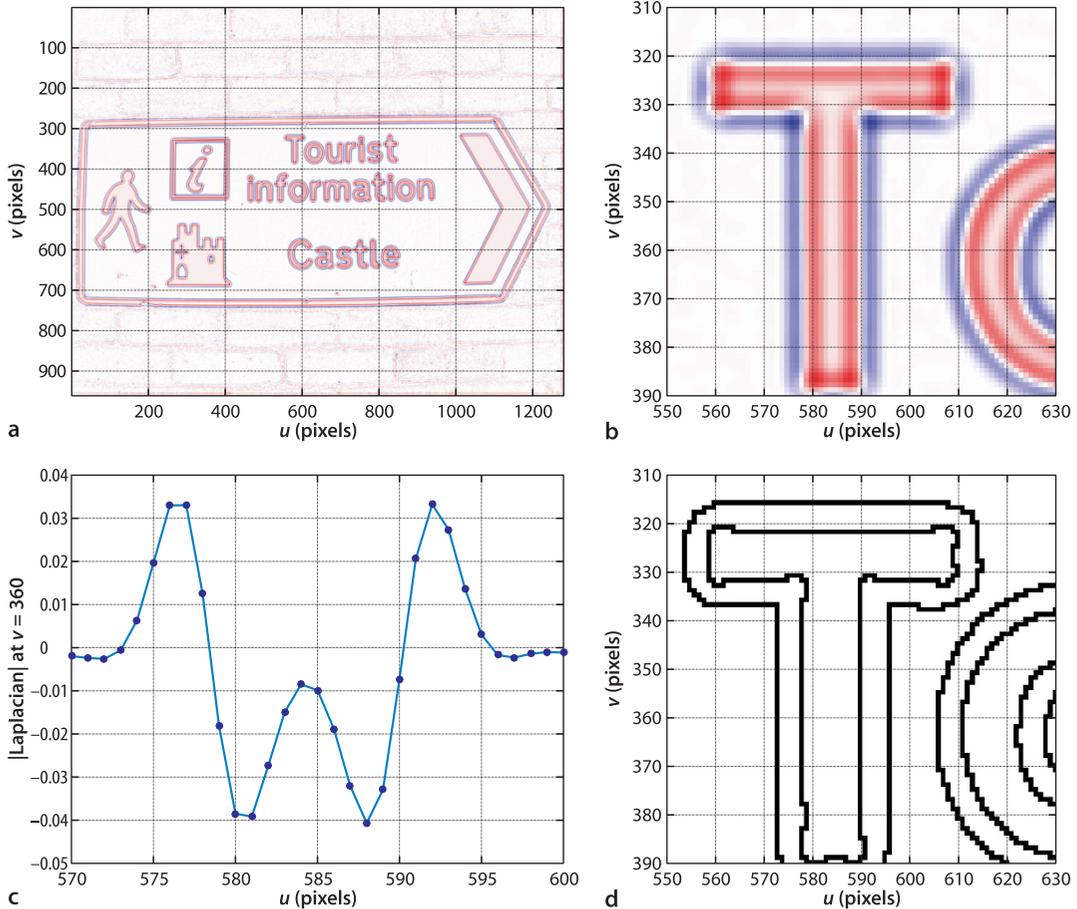
$$\mathbf{LoG}(u, v) = \frac{\partial^2 \mathbf{G}}{\partial u^2} + \frac{\partial^2 \mathbf{G}}{\partial v^2} \tag{12.7}$$

$$= \frac{1}{\pi\sigma^4}\left(\frac{u^2 + v^2}{2\sigma^2} - 1\right)e^{-\frac{u^2+v^2}{2\sigma^2}} \tag{12.8}$$

which is known as the Marr-Hildreth operator or the *Mexican hat kernel* and is shown in Fig. 12.14e.

We apply this kernel to our image by

```
>> lap = iconvolve( castle, klog(2) );
```

and the result is shown in Fig. 12.20a and b. The maximum gradient occurs where the second derivative is zero but a significant edge is a zero crossing from a strong positive value (blue) to a strong negative value (red). Consider the closeup view of the Laplacian of the letter T shown in Fig. 12.20b. We generate a horizontal cross-section of the stem of the letter T at $v = 360$

```
>> p = lap(360,570:600);
>> plot(570:600, p, '-o');
```

which is shown in Fig. 12.20c. We see that the zero values of the second derivative lies *between* the pixels. A zero crossing detector selects pixels adjacent to the zero crossing points

```
>> zc = zcross(lap);
```

and this is shown in Fig. 12.20d. We see that the edges appear twice. Referring again to Fig. 12.20c we observe a weak zero crossing in the interval $u \in [573, 574]$ and a much more definitive zero crossing in the interval $u \in [578, 579]$.

> A fundamental limitation of all edge detection approaches is that intensity edges do not necessarily delineate the boundaries of objects. The object may have poor contrast with the background which results in weak boundary edges. Conversely the object may have a stripe on it which is not its edge. Shadows frequently have very sharp edges but are not real objects. Object texture will result in a strong output from an edge detector at points not just on its boundary, as for example with the bricks in Fig. 12.16b.

12.5.2   Template Matching

In our discussion so far we have used kernels that represent mathematical functions such as the Gaussian and its derivative and its Laplacian. We have also considered the convolution kernel as a matrix, as an image and as a 3-dimensional surface as shown in Fig. 12.14. In this section we will consider that the kernel *is an image* or a part of an image which we refer to as a template. In template matching we wish to find which parts of the input image are most similar to the template.

Template matching is shown schematically in Fig. 12.21. Each pixel in the output image is given by

$$\mathbf{O}[u, v] = s(\mathbf{T}, \mathcal{W}), \quad \forall (u, v) \in \mathbf{I}$$

where $\mathbf{T}$ is the $w \times w$ template, the pattern of pixels we are looking for, with odd side length $w = 2h + 1$, and $\mathcal{W}$ is the $w \times w$ window centered at (u, v) in the input image. The function $s(\mathbf{I}_1, \mathbf{I}_2)$ is a scalar measure that describes the *similarity* of two equally sized images $\mathbf{I}_1$ and $\mathbf{I}_2$.

These measures can be augmented with a Gaussian weighting to deemphasize the differences that occur at the edges of the two windows.

A number of common similarity measures◄ are given in Table 12.1. The most intuitive are computed simply by computing the pixel-wise difference $T - \mathcal{W}$ and taking

David Marr (1945–1980) was a British neuroscientist and psychologist who synthesized results from psychology, artificial intelligence, and neurophysiology to create the discipline of Computational Neuroscience. He studied mathematics at Trinity College, Cambridge and his Ph.D. in physiology was concerned with modeling the function of the cerebellum. His key results were published in three journal papers between 1969 and 1971 and formed a theory of the function of the mammalian brain much of which remains relevant today. In 1973 he was a visiting scientist in the Artificial Intelligence Laboratory at MIT and later became a professor in the Department of Psychology. His attention shifted to the study of vision and in particular the so-called early visual system.

He died of leukemia at age 35 and his book *Vision: A computational investigation into the human representation and processing of visual information* (Marr 2010) was published after his death.
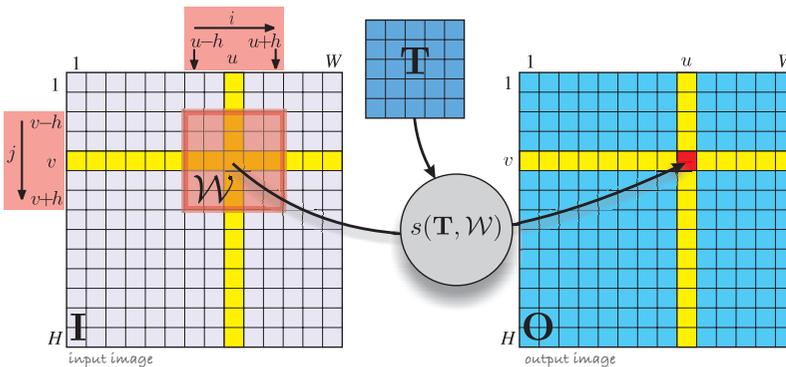
Fig. 12.21.
Spatial image processing operations. The *red shaded region* shows the window $\mathcal{W}$ that is the set of pixels used to compute the output pixel (shown in *red*)

the sum of the absolute differences (SAD) or the sum of the squared differences (SSD). These metrics are zero if the images are identical and increase with dissimilarity. It is not easy to say what value of the measure constitutes a poor match but a ranking of similarity measures can be used to determine the *best* match.

More complex measures such as normalized cross-correlation yield a score in the interval $[-1, +1]$ with $+1$ for identical regions. In practice a value greater than 0.8 is considered to be a good match. Normalized cross correlation is computationally more expensive – requiring multiplication, division and square root operations. Note that it is possible for the result to be undefined if the denominator is zero, which occurs if the elements of either I_1 or I_2 are identical.

If $I_2 \equiv I_1$ then it is easily shown that $SAD = SSD = 0$ and $NCC = 1$ indicating a perfect match. To illustrate we will use the Mona Lisa's eye as a 51×51 template

```
>> mona = iread('monalisa.png', 'double', 'grey');
>> T = mona(170:220, 245:295);
```

and evaluate the three common measures

```
>> sad(T, T)
ans =
     0
>> ssd(T, T)
ans =
     0
>> ncc(T, T)
ans =
     1
```

Now consider the case where the two images are of the same scene but one image is darker than the other – the illumination or the camera exposure has changed. In this case $I_2 = \alpha I_1$ and now

```
>> sad(T, T*0.9)
ans =
   111.1376
>> ssd(T, T*0.9)
ans =
     5.6492
```

Table 12.1.
Similarity measures for two equal sized image regions I_1 and I_2. The Z-prefix indicates that the measure accounts for the zero-offset or the difference in mean of the two images (Banks and Corke 2001). $\bar{I}_1$ and $\bar{I}_2$ are the mean of image regions I_1 and I_2 respectively. Toolbox functions are indicated in the last column

Sum of absolute differences		
SAD	$s = \Sigma_{(u,v) \in I_1} \left| I_1[u,v] - I_2[u,v] \right|$	sad
ZSAD	$s = \Sigma_{(u,v) \in I_1} \left| (I_1[u,v] - \bar{I}_1) - (I_2[u,v] - \bar{I}_2) \right|$	zsad
Sum of squared differences		
SSD	$s = \Sigma_{(u,v) \in I_1} \left(I_1[u,v] - I_2[u,v] \right)^2$	ssd
ZSSD	$s = \Sigma_{(u,v) \in I_1} \left((I_1[u,v] - \bar{I}_1) - (I_2[u,v] - \bar{I}_2) \right)^2$	zssd
Cross correlation		
NCC	$s = \dfrac{\Sigma_{(u,v) \in I_1} I_1[u,v]\, I_2[u,v]}{\sqrt{\Sigma_{(u,v) \in I_1} I_1^2[u,v]\ \Sigma_{(u,v) \in I_1} I_2^2[u,v]}}$	ncc
ZNCC	$s = \dfrac{\Sigma_{(u,v) \in I_1} (I_1[u,v] - \bar{I}_1)\ (I_2[u,v] - \bar{I}_2)}{\sqrt{\Sigma_{(u,v) \in I_1} (I_1[u,v] - \bar{I}_1)^2\ \Sigma_{(u,v) \in I_1} (I_2[u,v] - \bar{I}_2)^2}}$	zncc

these measure indicate a degree of dissimilarity. However the normalized cross-correlation

```
>> ncc(T, T*0.9)
ans =
    1
```

is invariant to the change in intensity.

Next consider that the pixel values have an offset◄ so that $I_2 = I_1 + \beta$ and we find that

This could be due to an incorrect black level setting. A camera's black level is the value of a pixel corresponding to no light and is often >0.

```
>> sad(T, T+0.1)
ans =
  260.1000
>> ssd(T, T+0.1)
ans =
   26.0100
>> ncc(T, T+0.1)
ans =
    0.9974
```

all measures now indicate a degree of dissimilarity. The problematic offset can be dealt with by first subtracting from each of **T** and $\mathcal{W}$ their mean value

```
>> zsad(T, T+0.1)
ans =
    3.5670e-12
>> zssd(T, T+0.1)
ans =
    4.8935e-27
>> zncc(T, T+0.1)
ans =
    1.0000
```

and these measures now all indicate a perfect match. The z-prefix denotes variants of the similarity measures described above that are invariant to intensity offset. Only the ZNCC measure

```
>> zncc(T, T*0.9+0.1)
ans =
    1.0000
```

is invariant to both gain and offset variation. All these methods will fail if the images have even a small change in relative rotation or scale.

Consider the problem from the well known children's book "Where's Wally" or "Where's Waldo" – the fun is trying to find Wally's face in a crowd

```
>> crowd = iread('wheres-wally.png', 'double');
>> idisp(crowd)
```

Fortunately we know roughly what he looks like and the template

```
>> wally = iread('wally.png', 'double');
>> idisp(wally)
```

was extracted from a different image and scaled so that the head is approximately the same width as other heads in the crowd scene (around 21 pixel wide).

The similarity of our template `wally` to every possible window location is computed by

```
>> S = isimilarity(wally, crowd, @zncc);
```

using the matching measure ZNCC. The result

```
>> idisp(S, 'colormap', 'jet', 'bar')
```

is shown in Fig. 12.22 and the pixel color indicates the ZNCC similarity as indicated by the color bar. We can see a number of spots of high similarity (white) which are candidate positions for Wally. The peak values, with respect to a local 3×3 window, are

```
>> [mx,p] = peak2(S, 1, 'npeaks', 5);
>> mx
mx =
    0.5258    0.5230    0.5222    0.5032    0.5023
```

in descending order. The second argument specifies the window half-width $h = 1$ and the third argument specifies the number of peaks to return. The largest value 0.5258 is the similarity of the strongest match found. These matches occur at the coordinates (u, v) given by the second return value `p` and we can highlight these points on the scene

```
>> idisp(crowd);
>> plot_circle(p, 30, 'edgecolor', 'g')
>> plot_point(p, 'sequence', 'bold', 'textsize', 24, 'textcolor', 'y')
```

using green circles that are numbered sequentially. The best match at (261, 377) is in fact the correct answer – we found Wally! It is interesting to look at the other highly ranked candidates. Numbers two and three at the bottom of the image are people also wearing baseball caps who look quite similar.
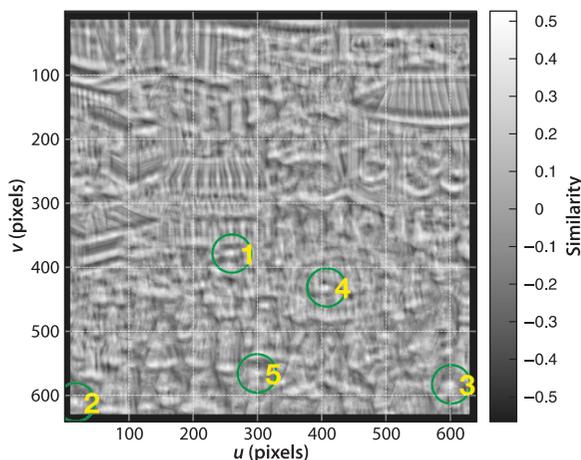
Fig. 12.22.
Similarity image S with top five Wally candidates marked. The *color bar* indicate the similarity scale. Note the border of indeterminate values where the template window falls off the edge of the input image

There are some important points to note from this example. The images have quite low resolution and the template is only 21×25 – it is a very crude likeness to Wally. The match is not a strong one – only 0.5258 compared to the maximum possible value of 1.0 and there are several contributing factors. The matching measure is not invariant to scale, that is, as the relative scale (zoom) changes the similarity score falls quite quickly. In practice perhaps a 10–20% change in scale between **T** and $\mathcal{W}$ can be tolerated. For this example the template was only approximately scaled. Secondly, not all Wallys are the same. Wally in the template is facing forward but the Wally we found in the image is looking to our left. Another problem is that the square template typically includes pixels from the background as well as the object of interest. As the object moves the background pixels may change, leading to a lower similarity score. This is known as the mixed pixel problem and is discussed in the next section. Ideally the template should bound the object of interest as tightly as possible. In practice another problem arises due to perspective distortion. A square pattern of pixels in the center of the image will appear keystone shaped at the edge of the image and thus will match less well with the square template.

A common problem with template matching is that false matches can occur. In the example above the second candidate had a similarity score only 0.5% lower than the first, the fifth candidate was only than 5% lower. In practice a number of rules are applied before a match is accepted: the similarity must exceed some threshold and the first candidate must exceed the second candidate by some factor to ensure there is no ambiguity.

Another approach is to bring more information to bear on the problem such as known motion of the camera or object. For example if we were tracking Wally from frame to frame in an image sequence then we would pick the best Wally closest to the previous location he was found. Alternatively we could create a motion model, typically a constant velocity model which assume he moves approximately the same distance and direction from frame to frame. In this way we could predict his future position and pick the Wally closest to that predicted position, or only search in the vicinity of the predicted position in order to reduce computation. We would also have to deal with practical difficulties such as Wally stopping, changing direction or being temporarily obscured.

12.5.2.1 Nonparameteric Local Transforms

Nonparametric similarity measures are more robust to the mixed pixel problem and we can apply a local transform to the image and template before matching. Two common transforms from this class are the census transform and the rank transform.

The census transform maps pixel values from a local region to an integer considered as a bit string – each bit corresponds to one pixel in the region as shown in Fig. 12.23. If a pixel is greater than the center pixel its corresponding bit is set to one, else it is zero. For a $w \times w$ window the string will be $w^2 - 1$ bits long. ◄ The two bit strings are compared using a Hamming distance which is the number of bits that are different.

For a 32-bit integer `uint32` this limits the window to 5×5 unless a sparse mapping is adopted (Humenberger et al. 2009). A 64-bit integer `uint64` supports a 7×7 window.

Fig. 12.23.
Example of census and rank transform for a 3×3 window. Pixels are marked *red* or *blue* if they are less than or greater than or equal to the center pixel respectively. These boolean values are then packed into a binary word, in the direction shown, from least significant bit upwards. The census value is 10101101_2 or decimal 173. The rank transform value is the total number of one bits and is 5
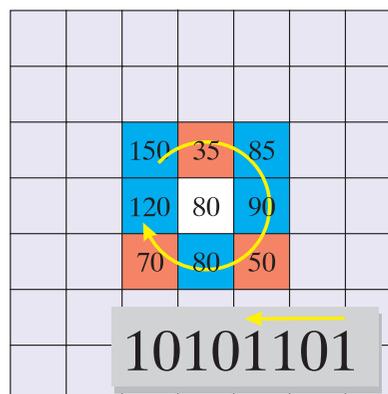
This can be computed by counting the number of set bits in the exclusive-or of the two bit strings. Thus very few arithmetic operations are required compared to the more conventional methods – no square roots or division – and such algorithms are amenable to implementation in special purpose hardware or FPGAs. Another advantage is that intensities are considered relative to the center pixel of the window making it invariant to overall changes in intensity or gradual intensity gradients.

The rank transform maps the pixel values in a local region to a scalar which is the number of elements in the region that are greater than the center pixel. This measure captures the *essence* of the region surrounding the center pixel, and like the census transform it is invariant to overall changes in intensity since it is based on local relative grey-scale values.

These transforms are typically used as a pre-processing step applied to each of the images before using a simple classical similarity measure such as SAD. The Toolbox function `isimilarity` supports these metrics using the `'census'` and `'rank'` options.

12.5.3   Nonlinear Operations

Another class of spatial operations is based on nonlinear functions of pixels within the window. For example

```
>> out = iwindow(mona, ones(7,7), 'var');
```

computes the variance of the pixels in *every* 7×7 window. The arguments specify the window size and the builtin MATLAB function `var`. The function is called with a 49×1 vector argument comprising the pixels in the window arranged as a column vector and the function's return value becomes the corresponding output pixel value. This operation acts as an edge detector since it has a low value for homogeneous regions irrespective of their brightness. It is however computationally expensive because the `var` function is called over 470 000 times. Any MATLAB function, builtin or your own M-file, that accepts a vector input and returns a scalar can be used in this way.

Rank filters sort the pixels within the window by value and return the specified element from the sorted list. The maximum value over a 5×5 window about each pixel is the first ranked pixel in the window

```
>> mx = irank(mona, 1, 2);
```

where the arguments are the rank and the window half-width $h = 2$. The median over a 5×5 window is the twelfth in rank

```
>> med = irank(mona, 12, 2);
```

and is useful as a filter to remove impulse-type noise and for the Mona Lisa image this significantly reduces the fine surface cracking. A more powerful demonstration is to add significant impulse noise to a copy of the Lena image

```
>> lena = iread('lena.pgm', 'double'); spotty = lena;
>> npix = prod(size(lena));
>> spotty(round(rand(5000,1)*(npix-1)+1)) = 0;
>> spotty(round(rand(5000,1)*(npix-1)+1)) = 1.0;
>> idisp(spotty)
```

and this is shown in Fig. 12.24a. We have set 5 000 random pixels to be zero, and another 5 000 random pixels to the maximum value. This type of noise is often referred to as impulse noise or salt and pepper noise. We apply a 3×3 median filter

```
>> idisp( irank(spotty, 5, 1) )
```

and the result shown in Fig. 12.24b is considerably improved. A similar effect could have been obtained by smoothing but that would tend to blur the image, median filtering has the advantage of preserving edges in the scene.
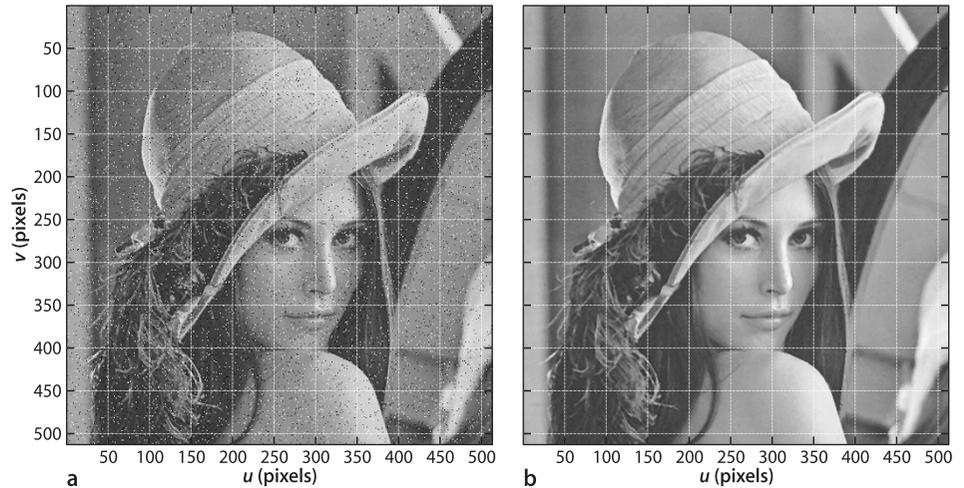
Fig. 12.24.
Median filter cleanup of impulse noise. **a** Noise corrupted image; **b** median filtered result

The third argument to `irank` can be a matrix instead of a scalar and this allows for some very powerful operations. For example

```
>> M = ones(3,3);
>> M(2,2) = 0
M =
     1    1    1
     1    0    1
     1    1    1
>> mxn = irank(lena, 1, M);
```

specifies the first in rank (maximum) over a *subset* of pixels from the window corresponding to the nonzero elements of `M`. In this case `M` specifies the eight neighboring pixels but not the center pixel. The result `mxn` is the maximum of the eight neighbors of each corresponding pixel in the input image. We can use this

```
>> idisp(lena > mxn)
```

to display all those points where the pixel value is greater than its local neighbors which performs nonlocal maxima suppression. These correspond to local maxima, or peaks if the image is considered as a surface. This mask matrix is very similar to a structuring element which we will meet in the next section.

12.6 Mathematical Morphology

Mathematical morphology is a class of nonlinear spatial operators shown schematically in Fig. 12.25. Each pixel in the output matrix is a function of a *subset* of pixels in a region surrounding the corresponding pixel in the input image

$$\mathbf{O}[u, v] = f\big(\mathbf{I}[u + i, v + j]\big), \quad \forall (i, j) \in \mathcal{S}, \quad \forall (u, v) \in \mathbf{I} \tag{12.9}$$

where $\mathcal{S}$ is the structuring element, an arbitrary small binary image. For implementation purposes this is embedded in a rectangular window with odd side lengths. The structuring element is similar to the convolution kernel discussed previously except that now it controls *which pixels* in the neighborhood the function $f(\cdot)$ is applied to – it specifies a subset of pixels within the window. The selected pixels are those for which the corresponding values of the structuring element are nonzero – these are shown in red in Fig. 12.25. Mathematical morphology, as its name implies, is concerned with the form or *shape* of objects in the image.
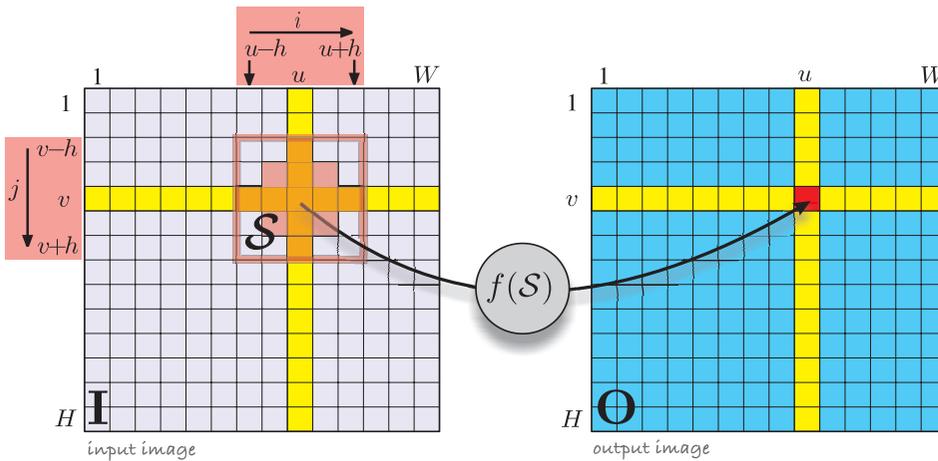
Fig. 12.25.
Morphological image processing operations. The operation is defined only for the selected elements (*red*) within the structuring element (*red outlined square*)

The easiest way to explain the concept is with a simple example, in this case a synthetic binary image created by the script

```
>> eg_morph1
>> idisp(im)
```

which is shown, repeated, down the first column of Fig. 12.26. The structuring element is shown in red at the end of each row. If we consider the top most row, the structuring element is a square

```
>> S = ones(5,5);
```

and is applied to the original image using the minimum operation

```
>> mn = imorph(im, S, 'min');
```

and the result is shown in the second column. For each pixel in the input image we take the *minimum* of all pixels in the 5 × 5 window. If *any* of those pixels are zero the resulting pixel will be zero. We can see this in animation by

```
>> morphdemo(im, S, 'min')
```

The result is dramatic – two objects have disappeared entirely and the two squares have become separated and smaller. The two objects that disappeared were not *consistent* with the shape of the structuring element. This is where the connection to morphology or shape comes in – only shapes that could *contain* the structuring element will be present in the output image.

The structuring element could define any shape: a circle, an annulus, a 5-pointed star, a line segment 20 pixels long at 30° to the horizontal, or the silhouette of a duck. Mathematical morphology allows very powerful shape-based filters to be created. The second row shows the results for a larger 7 × 7 structuring element which has resulted in the complete elimination of the small square and the further reduction of the large square. The third row shows the results for a structuring element which is a horizontal line segment 14 pixel wide, and the only remaining shapes are long horizontal lines.

The operation we just performed is often known as erosion since large objects are eroded and become smaller – in this case the 5 × 5 structuring element has caused two pixels▶ to be *shaved off* all the way around the perimeter of each shape. The small square, originally 5 × 5, is now only 1 × 1. If we repeated the operation the small square would disappear entirely, and the large square would be reduced even further.

The inverse operation is dilation which makes objects larger. In Fig. 12.26 we apply dilation to the second column results

The half width of the structuring element.

image   after erosion   after erosion then dilation

Fig. 12.26.
Mathematical morphology example. Pixels are either 0 (*grey*) or 1 (*white*). Each column corresponds to processing using the structuring element, shown at the end in *red*. The first column is the original image, the second column is after erosion by the structuring element, and the third column is after the second column is dilated by the structuring element
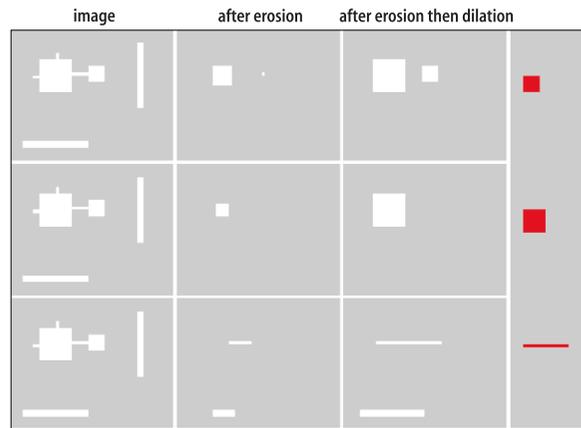
```
>> mx = imorph(mn, S, 'max');
```

and the results are shown in the third column. For each pixel in the input image we take the *maximum* of all pixels in the 5 × 5 window. If *any* of those neighbors is one the resulting pixel will be one. In this case we see that the two squares have returned to their original size, but the large square has lost its protrusions.

Morphological operations are often written in operator form. Erosion is

$$O = I \ominus S$$

where in Eq. 12.9 $f(\cdot) = \min(\cdot)$, and dilation is

$$O = I \oplus S$$

where in Eq. 12.9 $f(\cdot) = \max(\cdot)$. These operations are also known as Minkowski subtraction and addition respectively.

Erosion and dilation are related by

$$A \oplus B = \overline{\overline{A} \ominus B'}$$

where the bar denotes the logical complement of the pixel values, and the prime denotes reflection about the center pixel. Essentially this states that eroding the white pixels is the same as dilating the dark pixels and vice versa. For morphological operations

$$\left(A \oplus S_1\right) \oplus S_2 = A \oplus \left(S_1 \oplus S_2\right)$$
$$\left(A \ominus S_1\right) \ominus S_2 = A \ominus \left(S_1 \oplus S_2\right)$$

which means that successive erosion or dilation with a structuring element is equivalent to the application of a single larger structuring element, but the former is computationally cheaper. ◄ The shorthand functions

For example a 3 × 3 square structuring element applied twice is equivalent to 5 × 5 square structuring element. The former involves $2 \times (3 \times 3 \times N^2) = 18N^2$ operations whereas the later involves $5 \times 5 \times N^2 = 25N^2$ operations.
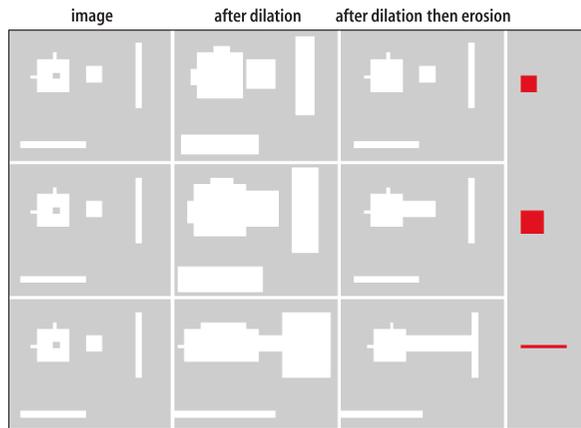
```
>> out = ierode(im, S);
>> out = idilate(im, S);
```

can be used instead of the low-level function `imorph`.

The sequence of operations, erosion then dilation, is known as opening since it opens up gaps. In operator form it is written as

$$I \circ S = (I \ominus S) \oplus S$$

Not only has the opening selected particular shapes but it has also *cleaned up* the image: the squares have been separated and the protrusions on the large square have been removed since they are not consistent with the shape of the structuring element.

In Fig. 12.27 we perform the operations in the opposite order, dilation then erosion. In the first row no shapes have been lost, they grew then shrank, and the large square still has its protrusions. The hole has been filled since it is not consistent with the shape of the structuring element. In the second row, the larger structuring element has caused the two squares to join together. This sequence of operations is referred to as closing since it closes gaps and is written in operator form as

$$I \bullet S = (I \oplus S) \ominus S$$

Note that in the bottom row the two line segments have remained attached to the edge, this is due to the default behavior in handling edge pixels.

Opening and closing▶ are implemented by the Toolbox functions `iopen` and `iclose` respectively. Unlike erosion and dilation repeated application of opening or closing is futile since those operations are idempotent

$$(I \circ S) \circ S = I \circ S$$
$$(I \bullet S) \bullet S = I \bullet S$$

These operations can also be applied to greyscale images to emphasize particular shaped objects in the scene prior to an operation like thresholding. A circular structuring element of radius R can be considered as a ball of radius R rolling on the intensity surface. Dilation, or the maximum operation, is the surface defined by the center of the ball rolling over the top of the input image intensity surface. Erosion, or the minimum operation, is the surface defined by the center of the ball rolling on the underside of the input image intensity surface.

These names make sense when considering what happens to white objects against a black background. For black objects the operations perform the inverse function.

12.6.1   Noise Removal

A common use of morphological opening is to remove noise in an image. The image

```
>> objects = iread('segmentation.png');
```

shown in Fig. 12.28a is a noisy binary image from the output of a rather poor object segmentation operation.▶ We wish to remove the dark pixels that do not belong to the objects and we wish to fill in the holes in the four dark rectangular objects.

Image segmentation and binarization is discussed in Sect. 13.1.1.

We choose a symmetric *circular* structuring element of radius 3

```
>> S = kcircle(3)
S =
     0     0     0     1     0     0     0
     0     1     1     1     1     1     0
     0     1     1     1     1     1     0
     1     1     1     1     1     1     1
     0     1     1     1     1     1     0
     0     1     1     1     1     1     0
     0     0     0     1     0     0     0
```

and apply a closing operation to fill the holes in the objects

```
>> closed = iclose(objects, S);
```

and the result is shown in Fig. 12.28b. The holes have been filled, but the noise pixels have grown to be small circles and some have agglomerated. We eliminate these by an opening operation

```
>> clean = iopen(closed, S);
```

and the result shown in Fig. 12.28c is a considerably cleaned up image. If we apply the operations in the inverse order, opening then closing

```
>> opened = iopen(objects, S);
>> closed = iclose(opened, S);
```

the results shown in Fig. 12.28d are much poorer. Although the opening has removed the isolated noise pixels it has removed large chunks of the targets which cannot be restored.
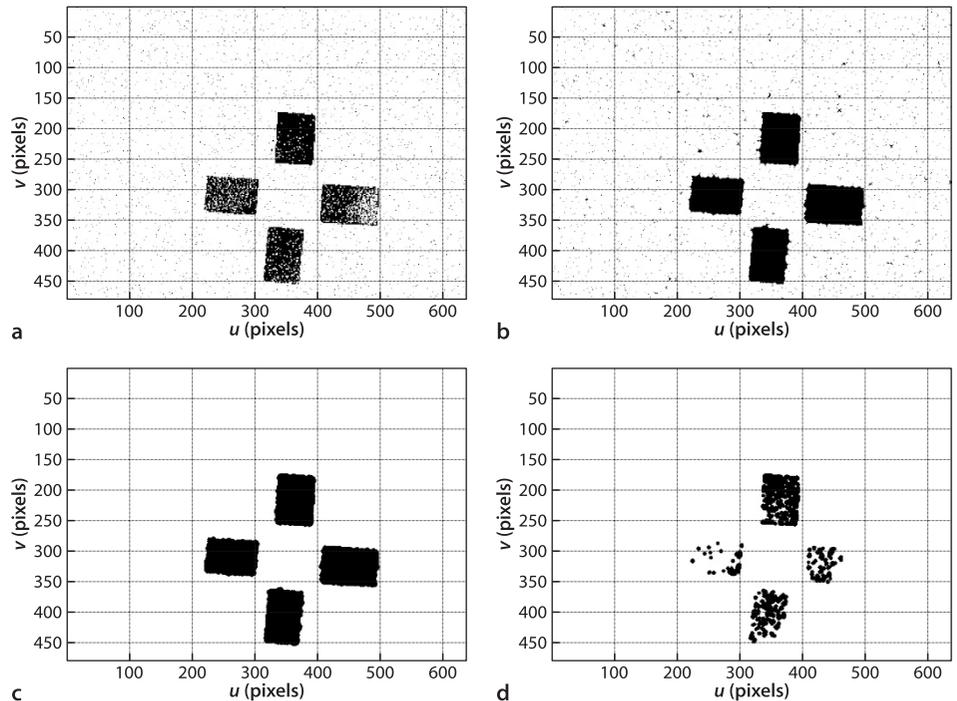
Fig. 12.28.
Morphological cleanup. **a** Original image, **b** original after opening, **c** opening then closing, **d** closing then opening. Structuring element is a *circle* of radius 3. Color map is inverted, set pixels are shown as *black*

12.6.2   Boundary Detection

The top-hat transform uses morphological operations to detect the edges of objects. Continuing the example from above, and using the image `clean` shown in Fig. 12.28c we compute its erosion using a circular structuring element

```
>> eroded = imorph(clean, kcircle(1), 'min');
```

The objects in this image are slightly smaller since the structuring element has caused one pixel to be *shaved off* the outside of each object. Subtracting the eroded image from the original

```
>> idisp(clean-eroded)
```

results in a layer of pixels around the edge of each object as shown in Fig. 12.29.

12.6.3   Hit or Miss Transform

The hit or miss transform uses a variation on the morphological structuring element. Its values are zero, one or *don't care* as shown in Fig. 12.30a. The zero and one pixels must exactly match the underlying image pixels in order for the result to be a one, as shown in Fig. 12.30b. If there is any mismatch of a one or zero as shown in Fig. 12.30c then the result will be zero. The Toolbox implementation is very similar to the morphological function, for example

```
out = hitormiss(image, S);
```

where the don't care elements of the structuring element are set to the special MATLAB value `NaN`.
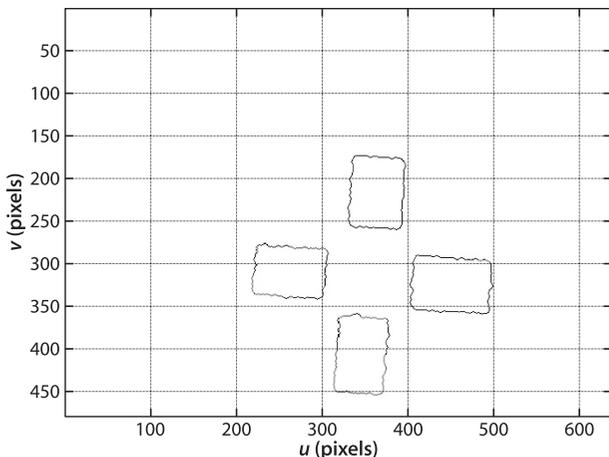

Fig. 12.29.
Boundary detection by morphological processing. Results are shown inverted, *white* is zero
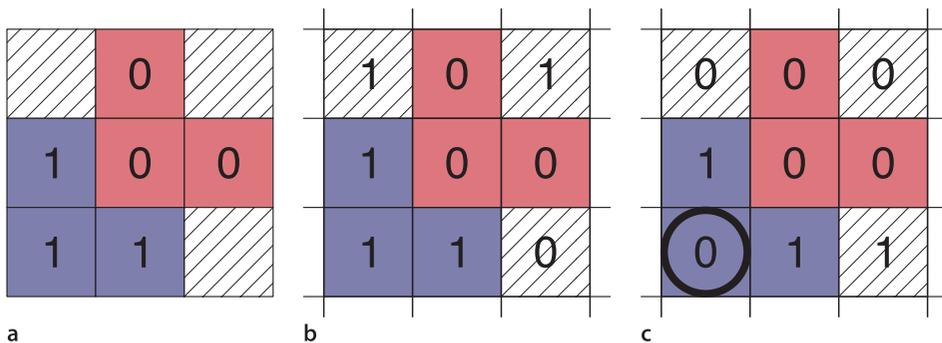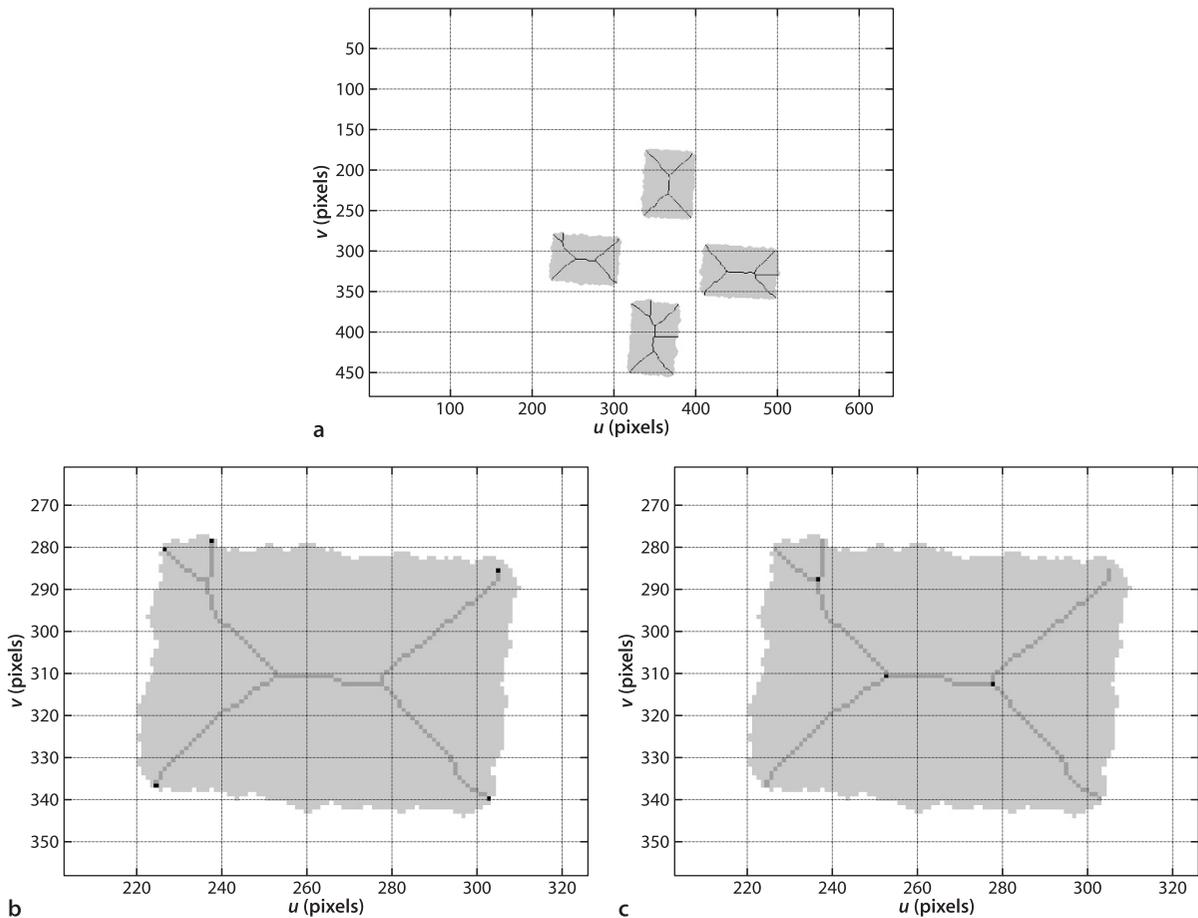

a   b   c

Fig. 12.30.
Hit or miss transform. **a** The structuring element has values of zero (*red*), one (*blue*), or don't care (*hatched*); **b** an example of a hit; **c** an example of a miss, the *pixel circled* is inconsistent with the structuring element

a

b

c

Fig. 12.31. Hit or miss transform operations. **a** Skeletonization; **b** end-point detection; **c** triple-point join detection. The images are shown inverted with the original binary image superimposed in *grey*. The end- and triplepoints are shown as *black pixels*

The hit or miss transform can be used iteratively with a sequence of structuring elements to perform complex operations such as skeletonization and linear feature detection. The skeleton of the objects is computed by

```
>> skeleton = ithin(clean);
```

and is shown in Fig. 12.31a. The lines are a single pixel wide and are the edges of a generalized Voronoi diagram – they delineate sets of pixels according to the shape boundary they are closest to. We can then find the endpoints of the skeleton

```
>> ends = iendpoint(skeleton);
```

and also the triplepoints

```
>> joins = itriplepoint(skeleton);
```

which are points at which three lines join. These are shown in Fig. 12.31b and c respectively.

For path planning in Sect. 5.2.1 we used a slow iterative wavefront approach to compute the distance transform. For this case a two pass algorithm can be used and if you have the MATLAB Image Processing Toolbox or VLFeat installed the faster functions `bwdist` or `vl_imdisttf` respectively will be used.

12.6.4   Distance Transform [examples/chamfer_match.m]

We discussed the distance transform in Sect. 5.2.1 for robot path planning. Given an occupancy grid it computed the distance of every free cell from the goal location. The distance transform we discuss here ◄ operates on a binary image and the output value, corresponding to every zero pixel in the input image, is the Euclidean distance to the nearest nonzero pixel.

Consider the problem of fitting a model to a shape in an image. We create the outline of a rotated square

```
>> im = testpattern('squares', 256, 256, 128);
>> im = irotate(im, -0.3);
>> edges = icanny(im) > 0;
```

which is shown in Fig. 12.32a and then compute the distance transform

```
>> dx = distancexform(edges, 'euclidean');
```

which is shown in Fig. 12.32b.

An initial estimate of the square is shown with a red line. The value of the distance transform at any point on this red square indicates how far away it is from the nearest
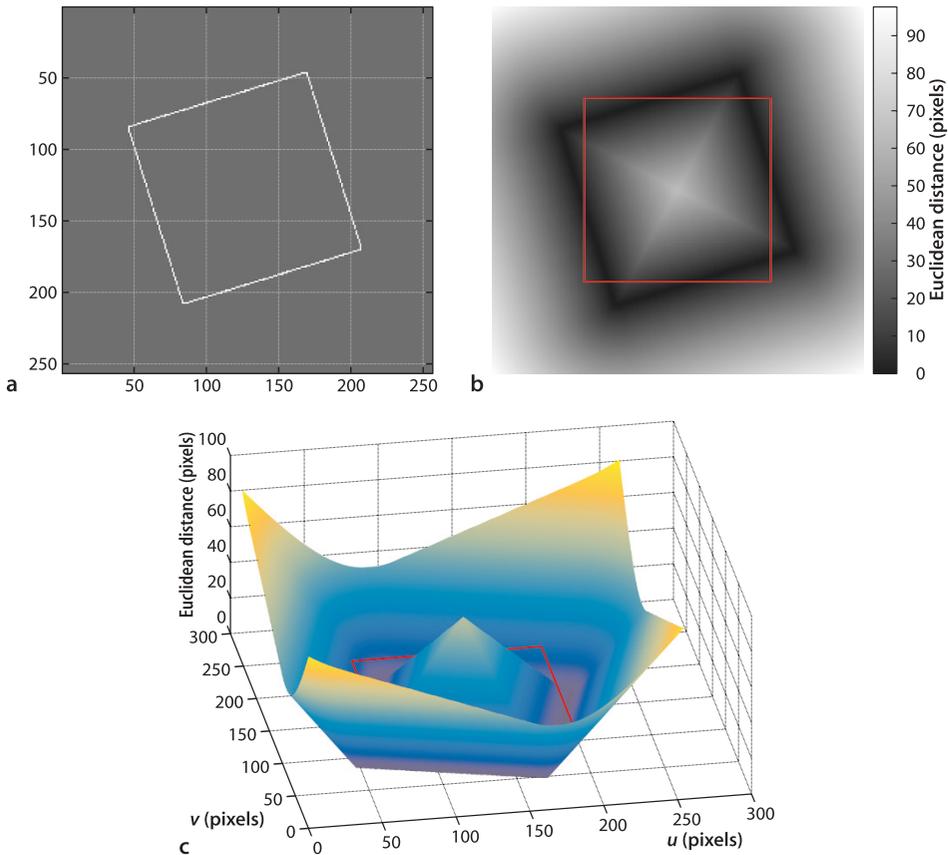
Fig. 12.32.
Distance transform. **a** Input binary image; **b** distance transformed input image with overlaid model square; **c** distance transform as a surface

The **distance transform** of a binary image has a value at each pixel equal to the distance from that pixel to the nearest nonzero pixel in the input image. The distance metric is typically either Euclidean (L_2 norm) or Manhattan distance (L_1 norm). It is zero for pixels that are nonzero in the input image. This transform is closely related to the **signed distance function** whose value at any point is the distance of that point to the nearest boundary of a shape, and is positive inside the shape and negative outside the shape. The figure shows the signed distance function for a unit circle, and has a value of zero, indicated by the red plane, at the object boundary. If we consider a shape to be defined by its signed distance transform then its zero contour defines the shape boundary.
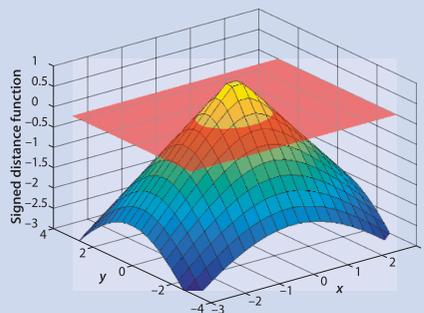
point on the original square. If we summed the distance transform for every point on the red square, or even just the vertices, we obtain a total distance measure which will only be zero when our model square overlays the original square. The total distance is a cost function which we can minimize using an optimization routine that adjusts the position, orientation and size of the square. Considering the distance transform as a 3-dimensional surface in Fig. 12.32c, our problem is analogous to dropping an extensible square hoop into the valley of the distance transform. Note that the distance transform only needs to be computed once, and during model fitting the cost function is simply a lookup of the computed distance. This is an example of chamfer matching and a full example, with optimization, is given in `examples/chamfer_match.m`.

12.7   Shape Changing

The final class of image processing operations that we will discuss are those that change the shape or size of an image.

12.7.1   Cropping

The simplest shape change of all is selecting a rectangular region from an image which is the familiar *cropping* operation. Consider the image

```
>> mona = iread('monalisa.png');
```

shown in Fig. 12.33a from which we interactively specify a region of interest or ROI

```
>> [eyes,roi] = iroi(mona);
>> idisp(eyes)
```

by clicking and dragging a selection box over the image. In this case we selected the eyes, and the corners of the selected region can be optionally returned and in this case was

```
>> roi
roi =
   239   359
   237   294
```

where the columns are the (u, v) coordinates for the top-left and bottom-right corners respectively. The rows are the u- and v-span respectively. The function can be used noninteractively by specifying a ROI

```
>> smile = iroi(mona, [265 342; 264 286]);
```

which in this case selects the Mona Lisa's smile shown in Fig. 12.33b.
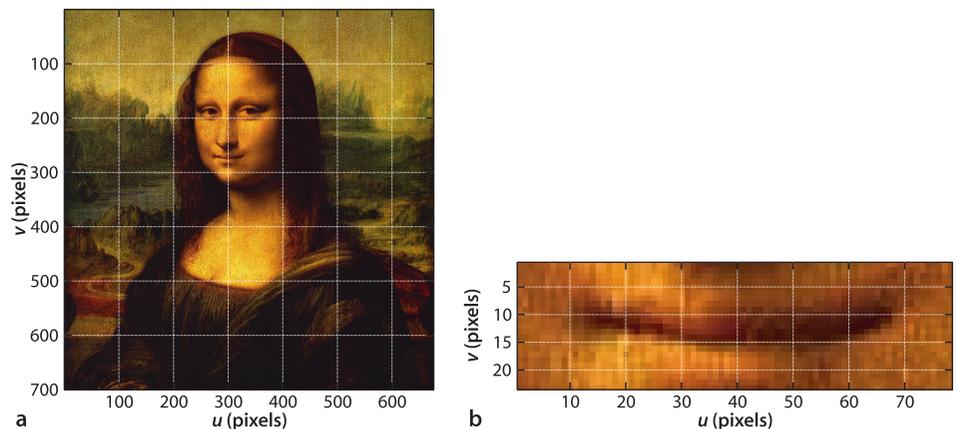
Fig. 12.33.
Example of region of interest or image cropping. **a** Original image, **b** selected region of interest

12.7.2   Image Resizing

Often we wish to reduce the dimensions of an image, perhaps because the large number of pixels results in long processing time or requires too much memory. We demonstrate this with a high-resolution image

```
>> roof = iread('roof.jpg', 'grey');
>> about(roof)
roof [uint8] : 1668x2009 (3351012 bytes)
```

which is shown in Fig. 12.34a. The simplest means to reduce image size is subsampling or decimation which selects every m^{th} pixel in the u- and v-direction, where $m \in \mathbb{Z}^+$ is the subsampling factor. For example with $m = 2$ an $N \times N$ image becomes an $N / 2 \times N / 2$ images which has one quarter the number of pixels of the original image.

For this example we will reduce the image size by a factor of seven in each direction

```
>> smaller = roof(1:7:end,1:7:end);
```

using standard MATLAB indexing syntax to select every seventh row and column. The result is shown is shown in Fig. 12.34b and we observe some pronounced curved lines on the roof which were not in the original image. These are artifacts of the sampling process. Subsampling reduces the spatial sampling rate of the image which can lead to spatial aliasing of high-frequency components due to texture or sharp edges. To ensure that the Shannon-Nyquist sampling theorem is satisfied an anti-aliasing low-

Fig. 12.34. Image scaling example. **a** Original image; **b** subsampled with $m = 7$, note the axis scaling; **c** subsampled with $m = 7$ after smoothing; **d** image **c** restored to original size by pixel replication
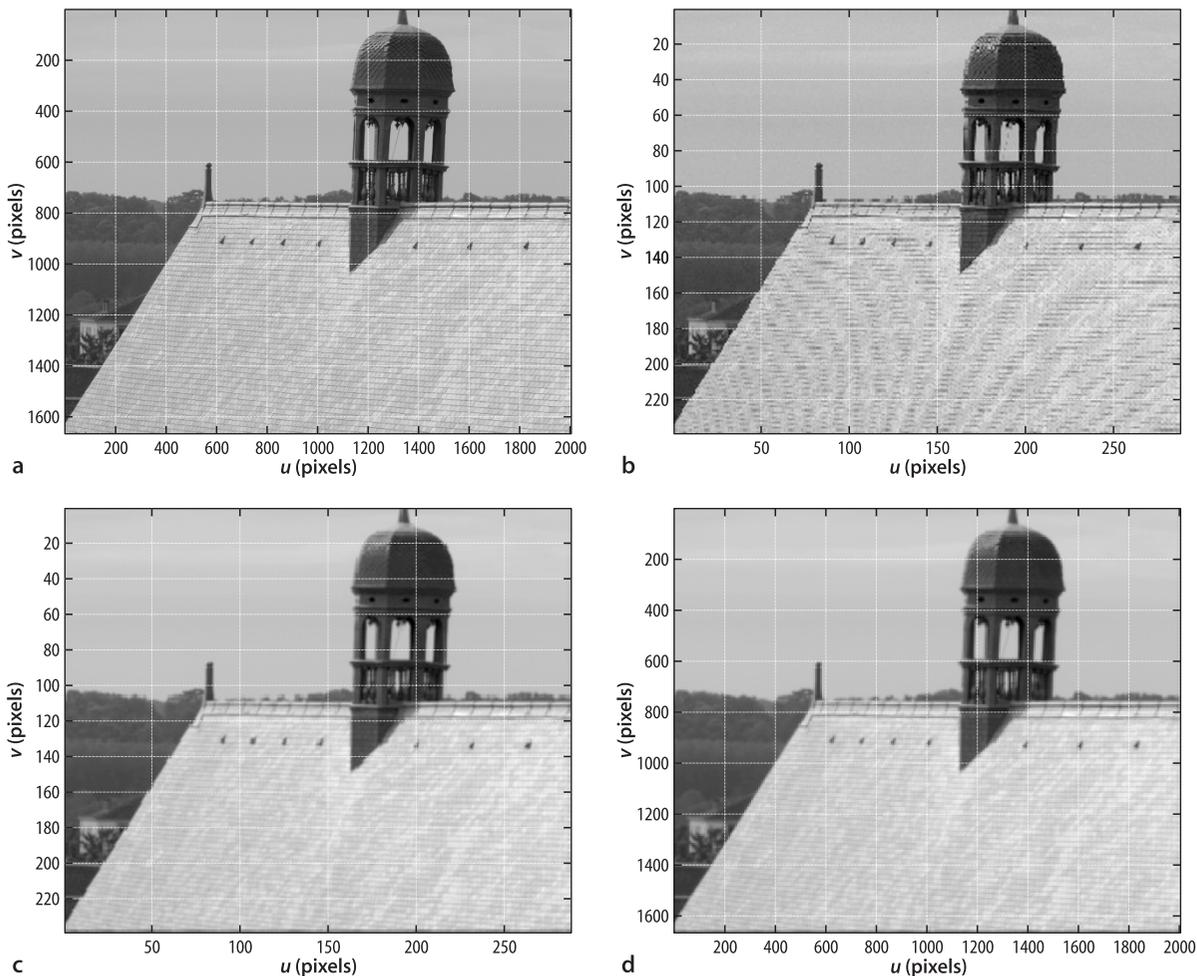
a

b

c

d

pass spatial filter must be applied to reduce the spatial bandwidth of the image before it is subsampled. ◄ This is another use for image blurring and the Gaussian kernel is a suitable low-pass filter for this purpose. The combined operation of smoothing and subsampling is implemented in the Toolbox by

```
>> smaller =  idecimate(roof, 7);
```

and the results for $m = 7$ are shown in Fig. 12.34c. We note that the curved line artifacts are no longer present.

The inverse operation is pixel replication, where each input pixel is replicated as an $m \times m$ tile in the output image

```
>> bigger = ireplicate( smaller, 7 );
```

which is shown in Fig. 12.34d and appears a little *blocky* along the edge of the roof and along the skyline. The decimation stage removed 98% of the pixels and restoring the image to its original size has not added any new information. However we could make the image easier on the eye by smoothing out the tile boundaries

```
>> smoother = ismooth( bigger, 4);
```

We can perform the same function using the Toolbox function `iscale` which scales an image by an arbitrary factor $m \in \mathbb{R}^+$ for example

```
>> smaller = iscale(lena, 0.1);
>> bigger = iscale(smaller, 10);
```

The second argument is the scale factor and if $m < 1$ the image will be reduced, and if $m > 1$ it will be expanded.

12.7.3   Image Pyramids

An important concept in computer vision, and one that we return to in the next chapter is scale space. The Toolbox function `ipyramid` returns a pyramidal decomposition of the input image

```
>> p = ipyramid( imono(mona) )
p =
  Columns 1 through 11
    [700x677 double] [350x339 double] ... [2x2 double]  [0.0302]
```

as a MATLAB cell array containing images at successively lower resolutions. Note that the last element is the 1×1 resolution version – a single dark grey pixel! These images are pasted into a composite image which is displayed in Fig. 12.35.
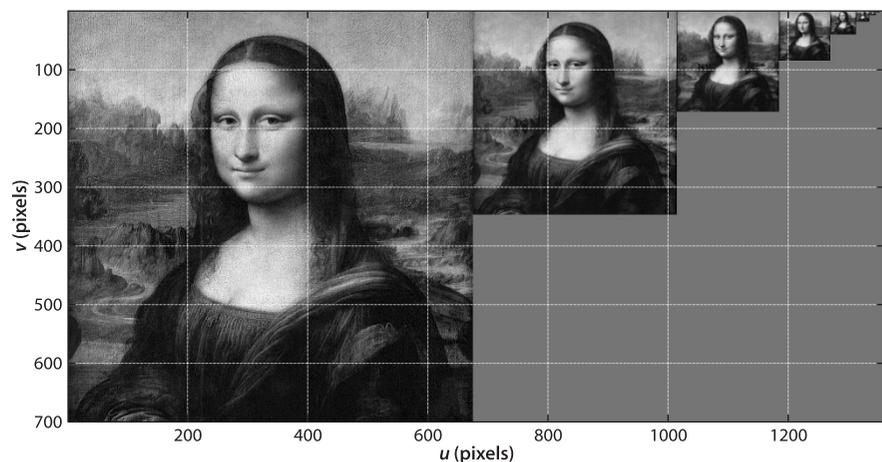
Fig. 12.35.
Image pyramid, a succession of images each half (by side length) the resolution of the one to the left

An image pyramid is the basis of many so-called coarse-to-fine strategies. Consider the problem of looking for a pattern of pixel values that represent some object of interest. The smallest image can be searched very quickly for the object since it comprises only a small number of pixels. The search is then refined using the next larger image but we now know which area of that larger image to search. The process is repeated until the object is located in the highest resolution image.

12.7.4   Image Warping

Image warping is a transformation of the pixel *coordinates* rather than the pixel values. Warping can be used to scale an image up or down in size, rotate an image or apply quite arbitrary shape changes. The coordinates of a pixel in the new view (u', v') are expressed as functions

$$u' = f_u(u, v), \quad v' = f_v(u, v) \tag{12.10}$$

of the coordinates in the original view.

Consider a simple example where the image is reduced in size by a factor of 4 in both directions and offset so that its origin, its top-left corner, is shifted to the coordinate (100, 200). We can express this concisely as

$$u' = u/4 + 100, \quad v' = v/4 + 200 \tag{12.11}$$

First we read the image and establish a pair of coordinate matrices▸ that span the domain of the input image, the set of all possible (u, v)

> The coordinate matrices are such that `U(u,v) = u` and `V(u,v) = v` and are a common construct in MATLAB see the documentation for `meshgrid`.

```
>> mona = iread('monalisa.png', 'double', 'grey');
>> [Ui,Vi] = imeshgrid(mona);
```

and another pair that span the domain of the output image, which we choose arbitrarily to be 400 × 400, the set of all possible (u', v')
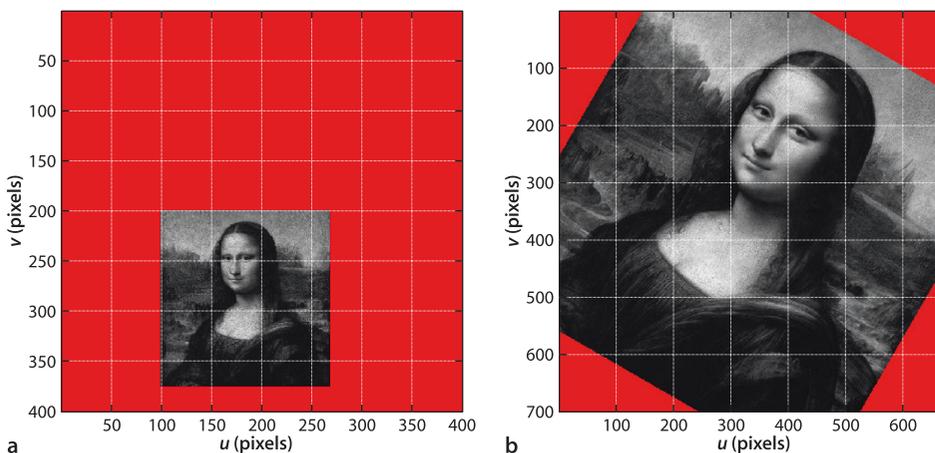
```
>> [Up,Vp] = imeshgrid(400, 400);
```

Now, for *every* pixel in the output image the corresponding coordinate in the input image is given by the inverse of the functions f_u and f_v. For our example the inverse of Eq. 12.11 is

$$u = 4(u' - 100), \quad v = 4(v' - 200) \tag{12.12}$$

which is implemented in matrix form in MATLAB as

```
>> U = 4*(Up-100); V = 4*(Vp-200);
```

a

b

Fig. 12.36.
Warped images. **a** Scaled and shifted; **b** rotated by 30° about its center. Pixels displayed as *red* were set to a value of `NaN` by `interp2` – they were not interpolated from any image pixels

We can now warp the input image using the MATLAB function `interp2`

```
>> little_mona = interp2(Ui, Vi, mona, U, V);
```

and the result is shown in Fig. 12.36a. Note that `interp2` requires a floating-point image.

Some subtle things happen under the hood. Firstly, while (u', v') are integer coordinates the input image coordinates (u, v) will not necessarily be integers. The pixel values must be interpolated◄ from neighboring pixels in the input image. Secondly, not all pixels in the output image have corresponding pixels in the input image as illustrated in Fig. 12.37. Fortunately for us `interp2` handles all these issues and pixels that do not exist in the input image are set to `NaN` in the output image which we have displayed as red. In case of mappings that are extremely distorted it may be that many adjacent output pixels map to the same input pixel and this leads to pixelation or *blockyness* in the output image.

Now let's try something a bit more ambitious and rotate the image by 30° into an output image of the same size as the input image

```
>> [Up,Vp] = imeshgrid(mona);
```

We want to rotate the image about its center but since the origin of the input image is the top-left corner we must first change the origin to the center, then rotate and then move the origin back to the top-left corner.◄ The warp equation is therefore

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \underbrace{\begin{pmatrix} \cos \pi/6 & -\sin \pi/6 \\ \sin \pi/6 & \cos \pi/6 \end{pmatrix}}_{R(\pi/6)} \begin{pmatrix} u - u_c \\ v - v_c \end{pmatrix} + \begin{pmatrix} u_c \\ v_c \end{pmatrix} \tag{12.13}$$

where (u_c, v_c) is the coordinate of the image center and $R(\frac{\pi}{6})$ is a rotation matrix in $\mathbf{SE}(2)$. This can be rearranged into the *inverse form* and implemented as

```
>> R = SO2(pi/6).R; uc = 256; vc = 256;
>> U = R(1,1)*(Up-uc) + R(2,1)*(Vp-vc) + uc;
>> V = R(1,2)*(Up-uc) + R(2,2)*(Vp-vc) + vc;
>> twisted_mona = interp2(Ui, Vi, mona, U, V);
```

and the result is shown in Fig. 12.36b. Note the direction of rotation – our definition of the x- and y-axes (parallel to the u- and v-axes respectively) is such that the z-axis is defined as being into the page making a clockwise rotation a positive angle. Also note that the corners of the original image have been lost, they fall outside the bounds of the output image.

The function `iscale` uses image warping to change image scale, and the function `irotate` uses warping to perform rotation. The example above could be achieved by

```
>> twisted_mona = irotate(mona, pi/6);
```

Finally we will revisit the lens distortion example from Sect. 11.2.4. The distorted image from the camera is the input image and will be warped to remove the distortion. We are in luck since the distortion model Eq. 11.13 is already in the inverse form. Recall that

$$u' = u + \delta_u$$
$$v' = v + \delta_v$$

where the distorted coordinates are denoted with a prime and δ_u and δ_v are functions of (u, v).

Different interpolation modes can be selected by a trailing argument to `interp2` but the default option is bilinear interpolation. A pixel at coordinate $(u + \delta_u, v + \delta_v)$ where $u, v \in \mathbb{Z}^+$ and $\delta_u, \delta_v \in [0, 1)$ is a linear combination of the pixels $(u, v), (u + 1, v), (u, v + 1)$ and $(u + 1, v + 1)$. The interpolation function acts as a weak anti-aliasing filter, but for very large reductions in scale the image should be smoothed first using a Gaussian kernel.

This is the application of a twist as discussed in Chap. 2.

Fig. 12.37.
Coordinate notation for image warping. The pixel (u', v') in the output image is sourced from the pixel at (u, v) in the input image as indicated by the *arrow*. The warped image is not necessarily polygonal, nor entirely contained within the output image
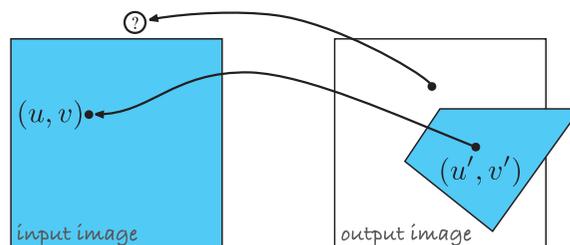
First we load the distorted image and build the coordinate matrices for the distorted and undistorted images

```
>> distorted = iread('Image18.tif', 'double');
>> [Ui,Vi] = imeshgrid(distorted);
>> Up = Ui; Vp = Vi;
```

and then load the results of the camera calibration

```
>> load  Bouguet
```

For readability we unpack the required parameters from the Calibration Toolbox variables `cc`, `fc` and `kc`

```
>> k = kc([1 2 5]); p = kc([3 4]);
>> u0 = cc(1); v0 = cc(2);
>> fpix_u = fc(1); fpix_v = fc(2);
```

for radial and tangential distortion vectors, principal point and focal length in pixels. Next we convert pixel coordinates to normalized image coordinates▶

In units of meters with respect to the camera's principal point.

```
>> u = (Up-u0) / fpix_u;
>> v = (Vp-v0) / fpix_v;
```

The radial distance of the pixels from the principal point is then

```
>> r = sqrt( u.^2 + v.^2 );
```

and the pixel coordinate errors due to distortion are

```
>> delta_u = u .* (k(1)*r.^2 + k(2)*r.^4 + k(3)*r.^6) + ...
   2*p(1)*u.*v + p(2)*(r.^2 + 2*u.^2);
>> delta_v = v .* (k(1)*r.^2 + k(2)*r.^4 + k(3)*r.^6) + ...
   p(1)*(r.^2 + 2*v.^2) + 2*p(2)*u.*v;
```

The distorted pixel coordinates in metric units are

```
>> ud = u + delta_u;  vd = v + delta_v;
```

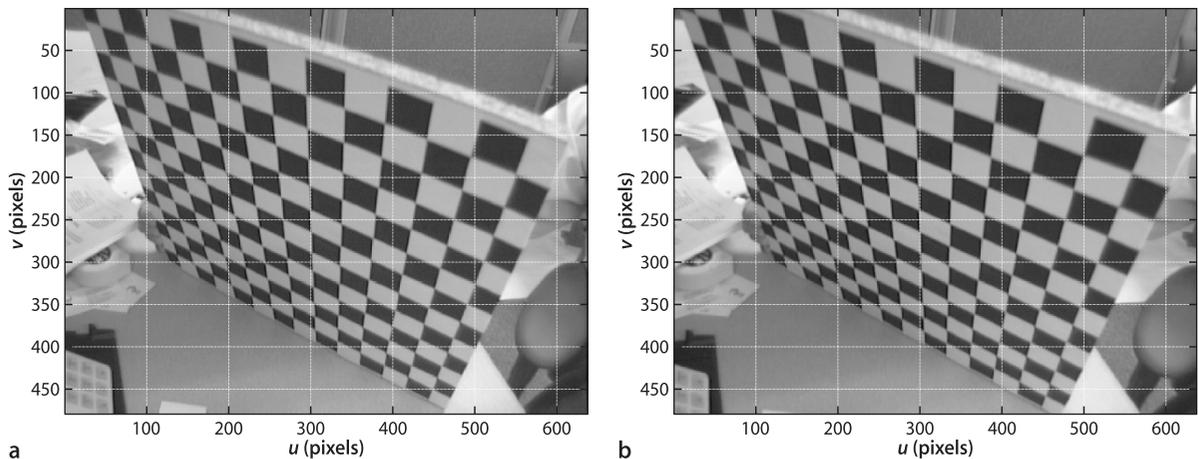which we convert back to pixel coordinates

```
>> U = ud * fpix_u + u0;
>> V = vd * fpix_v + v0;
```

and finally apply the warp

```
>> undistorted = interp2(Ui, Vi, distorted, U, V);
```

The results are shown in Fig. 12.38. The change is quite subtle, but is most pronounced at the edges and corners of the image where r is the greatest.

Fig. 12.38. Warping to undistort an image. **a** Original distorted image; **b** corrected image. Note that the top edge of the target has become a straight line (example from Bouguet's Camera Calibration Toolbox, image number 18)

a

b

12.8   Wrapping Up

In this chapter we learned how to acquire images from a variety of sources such as image files, movie files, video cameras and the internet, and load them into the MATLAB workspace. Once there we can treat them as matrices, the principal MATLAB datatype, and conveniently manipulate them. The elements of the image matrices can be integer, floating-point or logical values. Next we discussed many processing operations and a taxonomy of these is shown in Table 12.2. Operations on a single image include: unary arithmetic operations, type conversion, various color transformations and grey-level stretching; nonlinear operations such as histogram normalization and gamma encoding or decoding; and logical operations such as thresholding. We also discussed operations on pairs of images such as green screening, background estimation and moving object detection.

The largest and most diverse class of operations are spatial operators. We discussed convolution which can be used to smooth an image and to detect edges. Linear operations are defined by a kernel matrix which can be chosen to perform functions such as image smoothing (to reduce the effect of image noise or as a low-pass anti-aliasing filter prior to decimation) or for edge detection. Nonlinear spatial operations were used for template matching, computing rank statistics (including the median filter which eliminates impulse noise) and mathematical morphology which filters an image based on shape and can be used to cleanup binary images. A variant form, the hit or miss transform, can be used iteratively to perform functions such as skeletonization.

Finally we discussed shape changing operations such as regions of interest, scale changing and the problems that can arise due to aliasing, and generalized image warping which can be used for scaling, translation, rotation or undistorting an image. All these image processing techniques are the foundations of feature extraction algorithms that we discuss in the next chapter.

Further Reading

Image processing is a large field and this chapter has provided an introduction to many of the most useful techniques from a robotics perspective. More comprehensive coverage of the topics introduced here and others such as greyscale morphology, image restoration, wavelet and frequency domain methods, and image compression can be found in Szeliski (2011), Nixon and Aguado (2012), Forsyth and Ponce (2011) and Gonzalez and Woods (2011). Online information about computer vision is available through CVonline at http://homepages.inf.ed.ac.uk/rbf/CVonline, and the material in this chapter is covered under the section *Image Transformations and Filters*.

Edge detection is a subset of image processing but one with huge literature of its own. Forsyth and Ponce (2011) have a comprehensive introduction to edge detection and a useful discussion on the limitations of edge detection. Nixon and Aguado (2012) also cover phase congruency approaches to edge detection and compare various edge detectors. The Sobel kernel for edge detection was described in an unpublished 1968 publication from the Stanford AI lab by Irwin Sobel and Jerome Feldman: *A 3 × 3 Isotropic Gradient Operator for Image Processing*. The Canny edge detector was originally described in Canny (1983, 1987).

Nonparametric measures for image similarity became popular in the 1990s with with a number of key papers such as Zabih and Woodfill (1994), Banks and Corke (2001), Bhat and Nayar (2002). The application to real-time image processing systems using high-speed logic such as FPGAs has been explored by several groups (Corke et al. 1999; Woodfill and Von Herzen 1997).

Operation	RVC Toolbox	MATLAB
Monadic		
▪ Type conversion	`iint, idouble`	
▪ Unary functions		**`abs`**, **`sqrt`**, **`exp`**
▪ Unary operators		**`-, ~, .^2`**, `imcomplement`
▪ Image and constant		**`+, -, *, /, >, >=, <, <=`**, `imadd, imsubtract, immultiply, imdivide`
▪ Greyscale mapping	`igamm, istretch, inormhist`	`imadjust, stretchlim, histeq`
▪ Color	`imono, icolor, colorspace, tristim2cc`	`imapplymatrix,` **`rgb2hsv`**, **`hsv2rgb`**, `rgb2lab, lab2rgb, rgb2xyz, xyz2rgb, lab2xyz, xyz2lab`
Diadic		
▪ Binary operators		**`+, -, .*, ./, &, |`**, `imadd, imsubtract, immultiply, imdivide, imlincomb`
▪ Binary functions	`ipixswitch`	**`atan2`**, **`min`**, **`max`** `etc`
Spatial		
▪ Linear	`iconv, ismooth`	**`conv2`**, `imgaussfilt`
▪ Edge	`isobel, icanny`	`edge`
▪ Kernels	`kgauss, ksobel, kdgauss, klaplace, klog, kdog, kcircle`	`fspecial`
▪ Nonlinear	`irank, ivar, iwindow`	`ordfilt2, medfilt2, stdfilt`
▪ TemplateMatching	`isimilarity, sad, ssd, ncc, zsad, zssd, zncc`	`normxcorr2`
▪ Morphological	`imorph, ierode, idilate, iopen, iclose, hitormiss, ithin, iendpoint, itriplepoint`	`bwmorph, imerode, imdilate, imopen, imclose, bwhitmiss`
▪ Distance transform	`distancexform`[a]	`bwdist`
Shape changing		
▪ Scale	`idecimate, ireplicate, iscale`	`imresize`
▪ Rotate	`iscale`	`imrotate`
▪ Cropping	`iroi, isamesize, itrim, ipad`	`imcrop`
▪ Warping	`homtrans`	**`interp2,`** `imwarp, imtransform`
Creating images		
▪ from image file	`iread`	**`imread`**
▪ from camera	`VideoCamera`[a]	`videoinput`
▪ from web camera	`AxisWebCamera`	
▪ from movie file	`Movie`	**`VideoReader`**
▪ from Google maps	`EarthView`	
▪ from code	`testpattern, ipaste, kcircle, iline`	
General		
Display	`idisp`	**`imshow`**, `ImageViewer app`
Analysis	`ihist, iline`	
Writing image		**`imwrite`**

Notes: Bold means available in base MATLAB, others require additional MATLAB toolboxes. RVC Toolbox functions sometimes wrap base MATLAB functions.
[a] Function has extra functionality if MATLAB toolboxes are installed.

Mathematical morphology is another very large topic and we have only scraped the surface and important techniques such as greyscale morphology and watersheds have not been covered at all. The general image processing books mentioned above have useful discussion on this topic. Most of the specialist books in this field are now quite old but Shih (2009) is a good introduction and the book by Dougherty and Latufo (2003) has a more hands on tutorial approach.

The approach to computer vision covered in this book is often referred to as bottom-up processing. This chapter has been about *low-level* vision techniques which are operations on pixels. The next chapter is about *high-level* vision techniques where sets of pixels are grouped and then described so as to represent objects in the scene.

Sources of Image Data

All the images used in this part of the book are provided with the Toolbox in the images folder of the Machine Vision Toolbox.

There are thousands of online webcams as well as a number of sites that aggregate them and provide lists categorized by location, for example Opentopia, EarthCam and WatchThisCam. Most of these sites do not connect you directly to the web camera so the URL of the camera has to be dug out of the HTML page source. The root part of the URL (before the first single slash) is required for the AxisWebCamera class. Some of the content on these list pages can be rather dubious – so beware.

MATLAB Notes

Table 12.2 shows the image processing functions that have been discussed in this chapter and the equivalent functions from several toolboxes available from MathWorks: Image Processing Toolbox™, Image Acquisition Toolbox™ and Computer Vision System Toolbox™. There are many additional functions from these toolboxes that are not listed here. The RVC toolbox is open source and free, but its development is limited and the code is written for understanding rather than performance. In contrast the MathWorks' toolboxes are supported products and many have GPU support, can be used in Simulink or be used for automatic code generation. The companion to Gonzalez and Woods (2008) is their MATLAB based book from 2009 (Gonzalez et al. 2009) which provides a detailed coverage of image processing using MATLAB and includes functions that extend the IPT. These are provided for free as P-code format (no source or help available) or as M-files for purchase but are now quite dated.

The image processing search term at MATLAB CENTRAL http://www.mathworks.com/matlabcentral/fileexchange lists thousands of files.

General Software Tools

There are many high quality software tools for image and video manipulation outside the MATLAB environment. OpenCV at http://opencv.org is a mature open-source computer vision software project with over 2 500 algorithms, interfaces for C++, C, Python and Java and runs on Windows, Linux, Mac OS, iOS and Android. There are now several books about OpenCV and Kaehler and Bradski (2016) is the second edition of a popular book that provides a good introduction to the software and to computer vision in general.

ImageMagick http://www.imagemagick.org is a cross-platform collection of libraries and command-line tools for image format conversion (over 100 formats) and is useful for batch operations on large sets of images. For video manipulation FFmpeg http://www.ffmpeg.org is an excellent and comprehensive cross-platform tool. It supports conversion between video formats as well as videos to still images and vice versa.

Exercises

1. Become familiar with `idisp` for greyscale and color images. Explore pixel values in the image as well as the zoom, line and histogram buttons. Use `iroi` to extract the Mona Lisa's smile.
2. Look at the histogram of greyscale images that are under, well and over exposed. For a color image look at the histograms of the RGB color channels for scenes with different dominant colors. Combine real-time image capture with computation and display of the histogram.
3. Create two copies of a greyscale image into workspace variables `A` and `B`. Write code to time how long it takes to compute the difference of `A` and `B` using the MATLAB shorthand `A-B` or using two nested `for` loops. Use the functions `tic` and `toc` to perform the timing.
4. Grab some frames from the camera on your computer or from a movie file and display them.
5. Write a loop that grabs a frame from your camera and displays it. Add some effects to the image before display such as "negative image", thresholding, posterization, false color, edge filtering etc.
6. Given a scene with luminance of 800 nit and a camera with ISO of 1 000, $q = 0.7$ and f-number of 2.2 what exposure time is needed so that the average grey level of the 8-bit image is 150?
7. Images from space, page 367
   a) Obtain a map of the roads in your neighborhood. Use this to find a path between two locations, using the robot motion planners discussed in Chap. 5.
   b) For the images returned by the `EarthView` function write a function to convert pixel coordinate to latitude and longitude.
   c) Upload GPS track data from your phone and overlay it on a satellite image.
8. Motion detection
   a) Modify the Traffic example on page 375 and highlight the moving vehicles.
   b) Write a loop that performs background estimation using frames from your camera. What happens as you move objects in the scene, or let them sit there for a while? Explore the effect of changing the parameter σ.
   c) Combine concepts from motion detection and chroma-keying to put pixels from the camera where there is motion into the desert scene.
9. Convolution
   a) Compare the results of smoothing using a 21×21 uniform kernel and a Gaussian kernel. Can you observe the ringing artifact in the former?
   b) Why do we choose a smoothing kernel that sums to one?
   c) Compare the performance of the simple horizontal gradient kernel $K = (-0.5\ 0\ 0.5)$ with the Sobel kernel.
   d) Investigate filtering with the Gaussian kernel for different values of σ and kernel size.
   e) Create a 31×31 kernel to detect lines at 60 deg.
   f) Derive analytically the derivative of the Gaussian in the x-direction Eq. 12.4.
   g) Derive analytically the Laplacian of Gaussian Eq. 12.8.
   h) Derive analytically the difference of Gaussian from page 385.
   i) Show the difference between difference of Gaussian and derivative of Gaussian.
10. Show analytically the effect of an intensity scale error on the SSD and NCC similarity measures.
11. Template matching using the Mona Lisa image; convert it first to greyscale.
    a) Use `iroi` to select one of Mona Lisa's eyes as a template. The template should have odd dimensions.
    b) Use `isimilarity` to compute the similarity image. What is the best match and where does it occur? What is the similarity to the other eye? Where does the second best match occur and what is its similarity score?

c) Scale the intensity of the Mona Lisa image and investigate the effect on the peak similarity.

d) Add an offset to the intensity of the Mona Lisa image and investigate the effect on the peak similarity.

e) Repeat steps *(c)* and *(d)* for different similarity measures such as SAD, SSD, rank and census.

f) Scale the template size by different factors (use `iscale`) in the range 0.5 to 2.0 in steps of 0.05 and investigate the effect on the peak similarity. Plot peak similarity vs scale.

g) Repeat *(f)* for rotation of the template in the range -0.2 to 0.2 rad in steps of 0.05.

12. Perform the sub-sampling example on page 402 and examine aliasing artifacts around sharp edges and the regular texture of the roof tiles. What is the appropriate smoothing kernel width for a decimation by *M*?

13. Write a function to create Fig. 12.35 from the output of `ipyramid`.

14. Create a warp function that mimics your favorite funhouse mirror.

15. Warp the image to polar coordinates (r, θ) with respect to the center of the image, where the horizontal axis is r and the vertical axis is θ.