Image Processing with MATLAB

Supporting Material for
COMP27112

Dr. Tim Morris
Room 2.107
Kilburn Building
tim.morris@manchester.ac.uk

1 Introduction

MATLAB is being used as a platform for laboratory exercises and the problems classes in the Image Processing half of the Computer Graphics and Image Processing course unit. This handout describes the MATLAB development environment you will be using, you are expected to have read it and be familiar with it before attempting the Laboratory and Coursework Assignments.

MATLAB is a data analysis and visualisation tool designed to make matrix manipulation as simple as possible. In addition, it has powerful graphics capabilities and its own programming language. The basic MATLAB distribution can be expanded by adding a range of toolboxes, the one relevant to this course is the image-processing toolbox (IPT). The basic distribution and all of the currently available toolboxes are available in the labs. The basic distribution plus any installed toolboxes will provide a large selection of functions, invoked via a command line interface.

MATLAB's basic data structure is the matrix[1]. In MATLAB a single variable is a 1 x 1 matrix, a string is a 1 x n matrix of chars. An image is a n x m matrix of pixels. Pixels are explained in more detail below. Matrix operations are discussed in the appendix.

The handout summarises how the image processing operations discussed in lectures may be achieved in MATLAB, it summarises the MATLAB programming environment. Further help is available online, by either clicking on the "Help" menu item, or typing `helpbrowser` at the command prompt.

[1] A matrix is a rectangular array of objects of the same type. The matrix will have *r* rows and *c* columns, a matrix *element* is referenced as M[r,c], $M_{r,c}$ M_{rc} etc. Operations involving matrices are introduced as needed and summarised in the appendix.

2 MATLAB's Development Environment

MATLAB is started from within the Windows environment by clicking the icon that should be on the desktop. (MATLAB is also available for Linux and MacOS, but these environments are not supported in the laboratories, and these versions have their own licensing arrangements which are not covered by the University's licence.)

MATLAB's IDE has five components: the Command Window, the Workspace Browser, the Current Directory Window, the Command History Window and zero or more Figure Windows that are active only to display graphical objects.

The Command window is where commands and expressions are typed, and results are presented as appropriate.

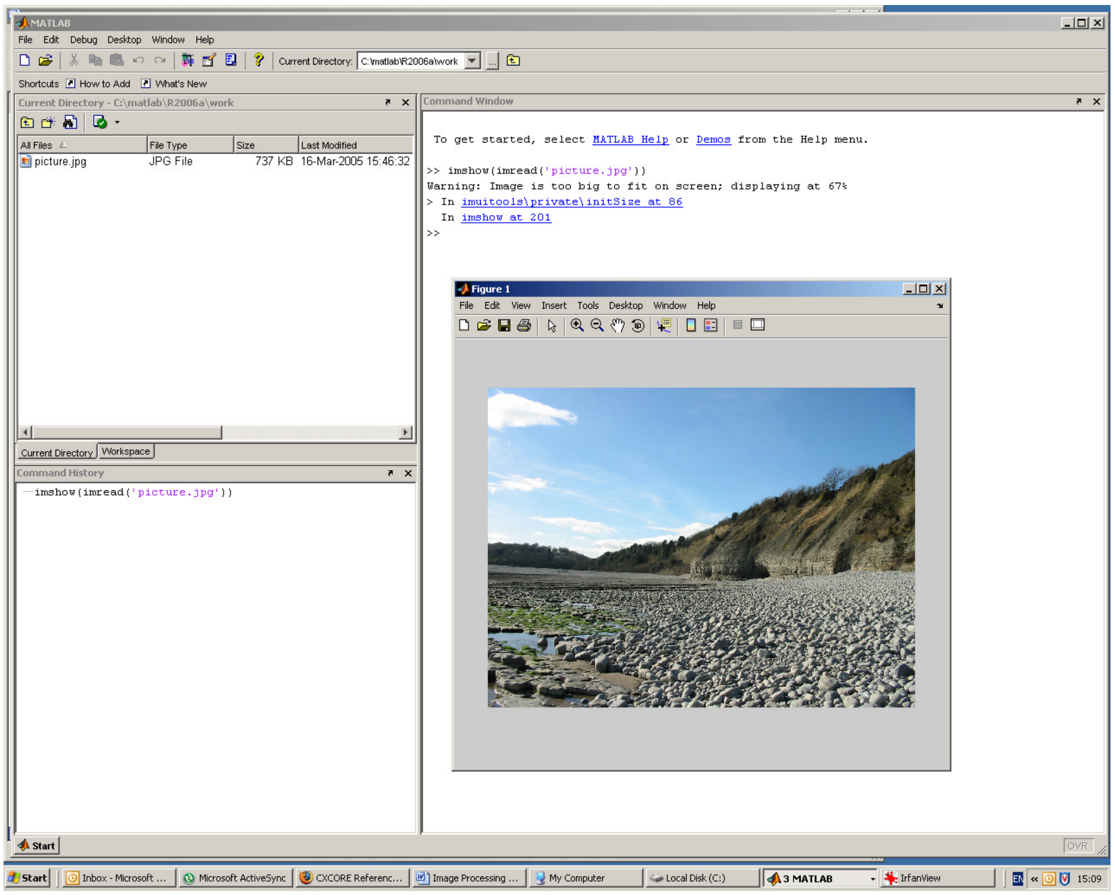
The workspace is the set of variables that have been created during a session. They are displayed in the Workspace Browser. Additional information about a variable is available there, some variables can also be edited.

The current directory window displays the contents of the current working directory and the paths of previous working directories. The working directory may be altered. MATLAB uses a search path to find files. The search path includes the current directory, all of the installed toolboxes plus any other paths that the user has added – via the Set Path dialogue accessed from the File menu.

The command history window gives a historical view of the current and previous sessions. Commands appearing here can be reexecuted.

MATLAB provides an editor for writing scripts. It is invoked by typing edit in the command window. Scripts are stored with the extension .m and are therefore also known as m-files. Some of the syntax is described in Appendix C

Figure windows are generated by MATLAB to display graphical objects.


MATLAB
File Edit Debug Desktop Window Help
Current Directory: C:\matlab\R2006a\work
Shortcuts How to Add What's New
Current Directory - C:\matlab\R2006a\work
Current Directory Workspace
Command History
Command Window
Figure 1
File Edit View Insert Tools Desktop Window Help
Start
OVR

All Files	File Type	Size	Last Modified
picture.jpg	JPG File	737 KB	16-Mar-2005 15:46:32

```
imshow(imread('picture.jpg'))
```

```
 To get started, select MATLAB Help or Demos from the Help menu.

>> imshow(imread('picture.jpg'))
Warning: Image is too big to fit on screen; displaying at 67%
> In imuitools\private\initSize at 86
  In imshow at 201
>>
```

Start
Inbox - Microsoft ...
Microsoft ActiveSync
CXCORE Referenc...
Image Processing ...
My Computer
Local Disk (C:)
3 MATLAB
IrfanView
15:09

Help on any MATLB command can be found in the Help Browser which is entered via the menu bar or by typing helpbrowser in the command window.

3 Image Representation

3.1 Image Format

An image is a rectangular array of values (pixels). Each pixel represents the measurement of some property of a scene measured over a finite area. The property could be many things, but we usually measure either the average brightness (one value) or the brightnesses of the image filtered through red, green and blue filters (three values). The values are normally represented by an eight bit integer, giving a range of 256 levels of brightness.

We talk about the *resolution* of an image: this is defined by the number of pixels and number of brightness values.

A raw image will take up a lot of storage space. Methods have been defined to compress the image by coding redundant data in a more efficient fashion, or by discarding the perceptually less significant information. MATLAB supports reading all of the common image formats. Image coding is not addressed in this course unit.

3.2 Image Loading and Displaying and Saving

An image is loaded into working memory using the command

```
>> f = imread('image file name');
```

The semicolon at the end of the command suppresses MATLAB output. Without it, MATLAB will execute the command and echo the results to the screen. We assign the image to the array f. If no path is specified, MATLAB will look for the image file in the current directory.

The image can be displayed using

```
>> imshow(f, G)
```

f is the image to be displayed, G defines the range of intensity levels used to display it. If it is omitted, the default value 256 is used. If the syntax [low, high] is used instead of G, values less than low are displayed as black, and ones greater than high are displayed as white. Finally, if low and high are left out, i.e. use [ ], low is set to the minimum value in the image and high to the maximum one, which is useful for automatically fixing the range of the image if it is very small or vary large.

Images are usually displayed in a figure window. If a second image is displayed it will overwrite the first, unless the figure function is used:

```
>> figure, imshow(f)
```

will generate a new figure window and display the image in it. Note that multiple functions may be called in a single line, provided they are separated by commas.

An image array may be written to file using:

```
>> imwrite(array name, 'file name')
```

The format of the file can be inferred from the file extension, or can be specified by a third argument. Certain file formats have additional arguments.

3.3 Image Information

Information about an image file may be found by

```
>> imfinfo filename
```

4 Quantisation

4.1 Grey Level Ranges

Images are normally captured with pixels in each channel being represented by eight bit integers. (This is partly for historical reasons – it has the convenience of being a basic memory unit, it allows for a suitable range of values to be represented, and many cameras could not capture data to any greater accuracy. Further, most displays are limited to eight bits per red, green and blue channel.) But there is no reason why pixels should be so limited, indeed, there are devices and applications that deliver and require higher resolution data. MATLAB supports the following data types.

Type	**Interpretation**	**Range**
uint8	unsigned 8-bit integer	[0, 255]
uint16	unsigned 16-bit integer	[0, 65535]
uint32	unsigned 32-bit integer	[0, 4294967295]
int8	signed 8-bit integer	[-128, 127]
int16	signed 16-bit integer	[-32768, 32767]
int32	signed 32-bit integer	[-2147483648, 2147483647]
single	single precision floating point number	[-10^{38}, 10^{38}]
double	double precision floating point number	[-10^{308}, 10^{308}]
char	character	(2 bytes per element)
logical	values are 0 or 1	1 byte per element

An image is usually interpreted as being one of: intensity, binary, indexed or RGB.

An intensity image's values represent brightnesses. A binary image's pixels have just two possible values. Indexed image's pixel values are treated as the index of a look-up table from which the "true" value is read. RGB images have three channels, representing intensities in ranges of wavelengths corresponding to red, green and blue illumination (other wavelength ranges and more of them are possible).

MATLAB provides functions for changing images from one type to another. The syntax is

```
>> B = data_class_name(A)
```

where data_class_name is one of the data types in the above table, e.g.

```
>> B = uint8(A)
```

will convert image A (of some type) into image B of unsigned 8-bit integers, with possible loss of resolution (values less than zero are fixed at zero, values greater than 255 are truncated to 255.)

Functions are provided for converting between image types, in which case the data is scaled to fit the new data type's range. These are defined to be:

Function	**Input Type**	**Output Type**
im2uint8	Logical, uint8, uint16, double	Unint8
im2uint16	Logical, uint8, uint16, double	Uint16
mat2gray	double	Double in range [0, 1]
im2double	Logical, uint8, uint16, double	Double
im2bw	Uint8, uint16, double	logical

4.2 Number of Pixels

Images come in all sizes, but are (almost) always rectangular. MATLAB gives several methods of accessing the elements of an array, i.e. the pixels of an image.

An element can be accessed directly: typing the array name at the prompt will return all the array elements (which could take a while), typing the array name followed by element indices in round brackets, will return that value.

Ranges of array elements can be accessed using colons.

```
>> A(first:last)
```

Will return the first to last elements inclusive of the one dimensional array A. Note that the indices start at one.

```
>> A(first : step : last)
```

Will return every step elements starting from first and finishing when last is reached or exceeded. Step could be negative, in which case you'd have to ensure that first was greater than last.

Naturally, this notation can be extended to access portions of an image. An image, f, could be flipped using

```
>> fp = f(end : -1 : 1, :);
```

The keyword end is used to signify the last index. Using the colon alone implies that all index values are traversed. This also indicates how multi-dimensional arrays are accessed.

Or a section of an image could be abstracted using

```
>> fc = f(top : bottom, left : right);
```

Or the image could be subsampled using

```
>> fs = f(1 : 2 : end, 1 : 2 : end);
```

4.2.1 A note on colour images.

If the input image is colour, these operations will return greyscale results. A colour image has three values per pixel, which are accessed using a third index.

```
>> A(x, y, 1:3)
```

Would return all three colour values of the pixel at *(x,y)*. A colour plane could be abstracted using

```
>> R = A(x, y, 1);
```

And similarly for G (last index = 2) and B.

5 Point Processing

Point processing operations manipulate individual pixel values, without regard to any neighbouring values. Two types of transforms can be identified, manipulating the two properties of a pixel: its value and position.

5.1 Value Manipulation

The fundamental value of a pixel is its brightness (in a monochrome image) or colour (in a multichannel image).

5.1.1 Pixel Scaling

Scaling of pixel values is achieved by multiplying by a constant. MATLAB provides a single function that achieves several effects

```
>> R = imadjust(A, [low_in, high_in], [low_out, high_out], gamma);
```

This takes the input range of values as specified and maps them to the output range that's specified. Values outside of the input range are clamped to the extremes of the output range (values below low_in are all mapped to low_out). The range values are expected to be in the interval [0, 1]. The function scales them to values appropriate to the image type before applying the scaling operation. Whilst low_in is expected to be less than high_in, the same is not true for low_out and high_out. The image can therefore be inverted.

The value of gamma specifies the shape of the mapped curve. Gamma = 1 gives a linear scaling, a smaller gamma gives a mapping that expands the scale at lower values, a larger gamma expands the upper range of the scale. This can make the contrast between darker or brighter tones more visible, respectively.

Omitting any of the parameters results in default values being assumed. The extremes of the ranges are used (0 and 1), or gamma = 1.

5.1.2 Histogram

The histogram of an image measures the number of pixels with a given grey or colour value. Histograms of colour images are not normally used, so will not be discussed here. The histogram of an image with L distinct intensity levels in the range [0, G] is defined as the function

$$h(r_k) = n_k$$

r_k is the k^{th} intensity level in the image, and n_k will be the number of pixels with grey value r_k. G will be 255 for a uint8 image, 65536 for uint16 and 1.0 for a double image. Since the lower index of MATLAB arrays is one, never zero, r_1 will correspond to intensity level 0, etc. For the integer valued images, $G = L\text{-}1$.

We often work with normalised histograms. A normalised histogram is obtained by dividing each element of $h(r_k)$ by the total number of pixels in the image (equal to the sum of histogram elements). Such a histogram is called the probability density function (pdf) and reflects the probability of a given intensity level occurring.

$$p(r_k) = n_k/n$$

MATLAB functions for computing and displaying the histogram and normalised histogram are:

```
>> h = imhist(A, b);
```

```
>> p = imhist(A, b)/numel(A);
```

b is the number of bins in the histogram. If it is omitted, 256 is assumed. The function numel(A) returns the number of elements in the argument, in this case the number of pixels in A.

Additional functions are defined for displaying the data in different forms: notably as a bar graph and for adding axes.

5.1.3 Histogram Equalisation

It is possible to manipulate the grey scale to achieve different effects. One of these effects to known as histogram equalisation. The aim is to transform the grey scale such that the pdf of the output image is uniform. Some authors claim that this improves the appearance of the image. The transformation is achieved using:

```
>> h = histeq(A, nlev);
```

Where nlev is the number of levels in the output image, h. Its default value is 64.

5.1.4 Thresholding

This is a simple method of differentiating between an object and the background, which works provided they are of different intensities. A *threshold* value is defined. Intensities greater than this are set to one value, intensities less than to another (1 or max, and 0 or min are often used). Whilst a threshold can be decided manually, it is better to use the image data to compute one.

Gonzalez and Woods suggested this method:

1. Choose a threshold arbitrarily.
2. Threshold the image using it.
3. Compute the mean grey value of the pixels with intensities above and below the threshold, and then compute the average of these two values.
4. Use the new value to rethreshold the image.
5. Repeat steps 3 and 4 until the threshold changes by an insignificant amount.

Otsu suggested a method than minimised the between class variance. This is the method provided by the MATLAB toolbox:

```
>> T = graythresh(A);
```

T is a normalised value between 0.0 and 1.0; it should be scaled to the proper range before using it. The conversion function im2bw will then return the thresholded image.

5.1.5 Colour Transforms

Multiple methods of representing colour data exist. Whilst RGB is most widely used for capture and display, it is not always the best for image processing, since it is a perceptually non-uniform representation. This means that if we change the RGB values by a fixed amount, the *observed* difference depends on the original RGB values. One way of observing this is to mix the output of standardised coloured lights to generate a colour, then alter the brightness of the input until an observer just notices a change in the light's colour. The original colour and the colour of the just noticeable difference can be plotted. By making measurements systematically over the whole colour space, we can generate a MacAdam diagram. The points represent the original colour, the ellipses the just noticeable difference contours.

It is also possible to categorise colour spaces as being device dependent or device independent. Device dependent spaces are used in the broadcast and printing industry, largely for convenience. The most widely used spaces are YIQ, YC_rC_b and HSV. Conversion between the spaces is by using simple functions. E.g.

```
>> YIQ = rgb2ntsc(RGB);
```

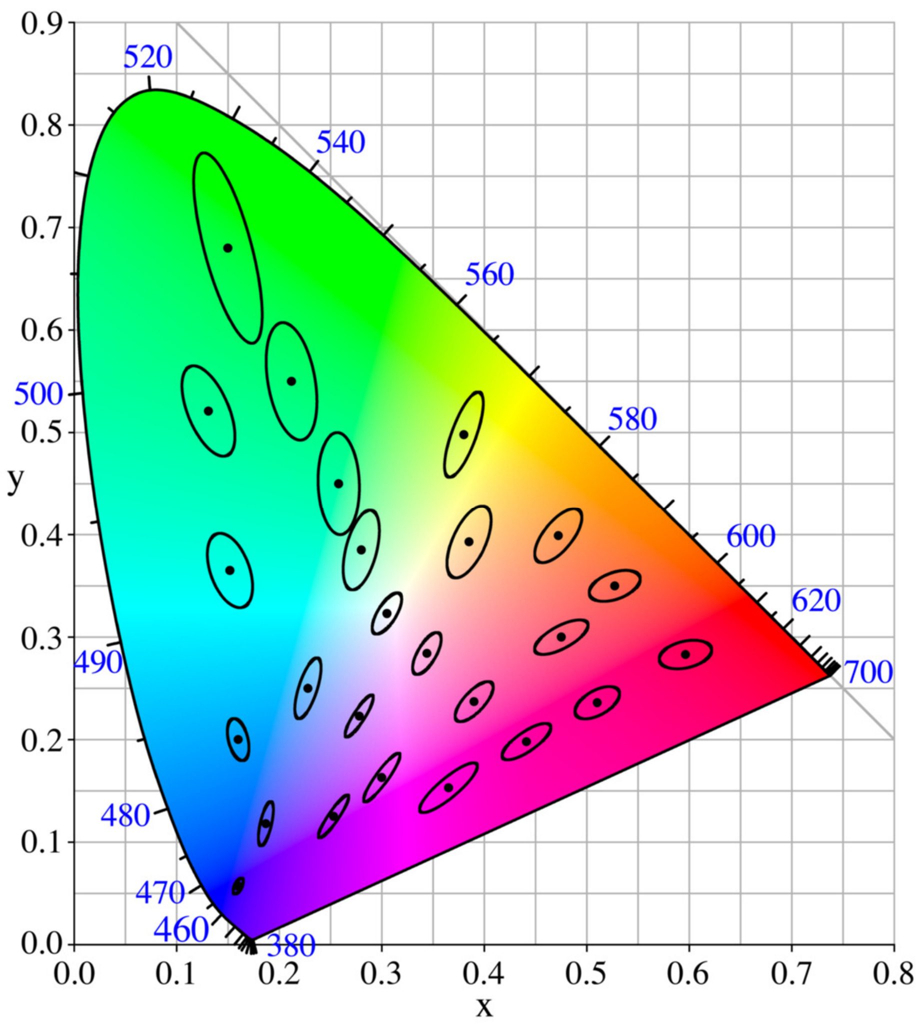
Device independent spaces are used because the device dependent spaces include subjective definitions. The CIE defined a standardised colour space in 1931. It specifies three colour sources, called X, Y and Z. All visible colours can be generated by a linear combination of these. The X, Y and Z values can be normalised, to sum to 1. The colours represented by the normalised x and y values can be plotted – as in the MacAdam diagram. Conversion of data between colour spaces is a two stage process. A colour transformation structure is first defined, e.g. to convert from RGB to XYZ:

```
>> C = makecform('srgb2xyz');
```

And then perform the conversion

```
>> Ixyz = applycform(Irgb, C);
```

The data types used to represent the data may alter. Other colour spaces have been defined, each attempting to make the colour space more perceptually uniform.



MacAdam Diagram
The horizontal and vertical axes represent the normalised x and y values (obviously the third component is unnecessary as z = 1 – x – y). Colours can only exist below the x + y = 1 line, and only some of these points represent visible colours. The colours of the rainbow appear around the outside, the blue figures are the wavelengths of the equivalent pure illumination.

5.2 Co-ordinate Manipulation

The result of co-ordinate manipulation is to distort an image. For example, in creating panoramic images, or correcting images for lens distortion in order to make measurements. Manipulation has two stages – computing the pixels' new co-ordinates and resampling, or interpolation.

5.2.1 Transform

There are two classes of transform: affine and non-linear. Affine transforms are achieved by inserting suitable values into a transform matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

One or other axis can be scaled, the image may be rotated or sheared, depending on the values of *a* to *i*. The transform is realised in MATLAB via a tform structure. A tform structure is created by the function:

```
>> tform = maketform(transform_type, transform_parameters);
```

transform_type will be one of affine, projective, box, composite or custom. The transform_parameters will be dependent on the transform type, in this section we are just interested in the affine transforms for which the parameters are the nine elements of the matrix. So we could set up a non-uniform scaling transform by

```
>> T = [2 0 0; 0 3 0; 0 0 1];
>> tform = maketform('affine', T);
```

Non-linear transforms are usually used to distort the image to correct for the distortions introduced by the imaging system.

$$x' = x(1 + a_1 x + \ldots)$$
$$y' = y(1 + b_1 y + \ldots)$$

5.2.2 Interpolation

It would seem natural to take a pixel in the input image and compute its location in the distorted image. But, the values of the affine transform parameters, or the distortion equations are generally non-integer. So the computed co-ordinates may be non-integer. Rounding errors can result in some output pixels not being populated. Therefore, the distortion is performed in reverse. We use the inverse of the transformation to compute the source of each pixel in the distorted image. The source pixel may well have non-integer co-ordinates, but we can estimate its value by taking the nearest pixel, or interpolating over the nearest pixels.

Forward transforms (the type we do not wish to do) are achieved by using tformfwd. tforminv or imtransform (a better choice) compute the inverse:

```
>> res = tforminv(src, tform, interp);
```

interp is a simple flag that specifies how the interpolation is to be performed.

Nearest	uses the nearest neighbour pixel.
Bilinear	(the default) uses a weighted average of the nearest 2 by 2 pixel neighbourhood.
Bicubic	uses a weighted average of the nearest 4 by 4 pixel neighbourhood.

6 Region Processing

Region processing operations compute the value of an output pixel using a set of the pixel's neighbours. Various sizes and shapes of neighbourhoods are used, ranging in complexity from the simplest 8 nearest neighbours upwards. The methods used to combine pixels' values also vary: some operations using weighted sums of pixels, others select the output using more complex logical operations.

Whilst it is possible to categorise the operations on the type of input, how the output is computed, etc, it is more instructive to look at *what* can be achieved using these methods. Two objectives can be achieved by manipulating pixels in a region: smoothing the data and sharpening it. Both can use a technique called convolution.

Convolution is defined mathematically by

$$c[i,j] = \sum_{k=-m}^{k=m} \sum_{l=-n}^{l=n} t[k,l]d[i-k, j-l]$$

Where *t[k,l]* is a template, alternatively known as a kernel, and *d[i,j]* is the input data. Conceptually, we place the template onto the image, multiply the overlapping values and sum them. This is done for all possible location in the image, obviously the edges where the temple extends outside the image deserve special attention. We often use a square template ($m = n$).

A similar operation to convolution is correlation, defined as:

$$c[i,j] = \sum_{k=-m}^{k=m} \sum_{l=-n}^{l=n} t[k,l]d[i+k, j+l]$$

Conceptually, the two operations are the same if we rotate the template by 180°. In fact, all of the templates we will be looking at are either symmetrical or anti-symmetrical under this rotation. This means that correlation and convolution will either give the same result or the same magnitude result (but different sign).

A simple method of understanding the effect of convolution/correlation with a given kernel, is that it measures the similarity between the image and the kernel. This will become more obvious when we look at specific kernels.

MATLAB provides a simple syntax for filtering:

```
>> dst = imfilter(src, mask, filtering_mode, boundary_options, size_options);
```

Where dst and src are the output and input images respectively. mask is the filtering mask aka kernel, template etc. filtering_mode specifies whether convolution ('conv') or correlation ('corr') is to be used. boundary_options specifies how the data near the image border is to be treated:

P	the boundary is padded with a value. The is the default mode, padding with 0
'replicate'	the image is expanded by copying the values in the outermost pixels
'symmetric'	the image is expanded by mirror-reflecting across a border
'circular'	the image is treated as a periodic 2-D function, values wraparound

size_options specifies how the destination image's size is affected by the padding: 'full' implies the output will be the same size as the expanded image; 'same' implies the image will be the size of the input image, this implies that the border pixels (the pixels at which the kernel has some points that fall outside the source image) are not processed.

A final point about convolution/correlation with a square kernel is that the same effect can be generated by two convolutions with two appropriately defined one-dimensional kernels. A 2-D convolution is replaced by two 1-D convolutions. This has a saving in processing, since the 2-D convolution requires about n^2 multiplications and additions, the 1-D case requires about 2n such operations.

6.1 High Pass - Edge Detection

An edge is defined as a significant, local change in image intensity. The simplest (ideal) edge is a line (the edge) separating uniform regions of different intensity. In practice, this is never observed as noise superimposes random fluctuations on the image data, and imperfect optics and digitisation blur the image. The ideal edge is therefore corrupted and we see a rather more gradual change in intensity from one region to the other. The problem of edge detection is to locate the edge in this data (if it is possible, and if it makes sense).

If an edge is a discontinuity in intensity, there must be a region around it where the intensity changes by large amounts over small distances, i.e. the gradient is high. The gradient can be estimated by measuring the differences in intensity over small distances and dividing by the distance – this is the digital equivalent of differentiation. Rather than take differences between individual pixels, local averages are computed and compared as this is less susceptible to the noise that is present.

An edge can take any orientation within an image. Differentiation can only be performed in directions parallel to the two axes. The edge strength and orientation can be estimated as described in Appendix B.

Once the convolution has been performed and edge strengths estimated, we obtain a grey scale image whose intensity is proportional to the likelihood of a pixel being on an edge. But if the edge has been smeared out, where is the "true" edge? Non-maximal suppression is used to remove pixels that are not located on the position of maximum slope: we can track across an edge using the edge orientation information and locate the pixel with the greatest edge strength.

6.1.1 Prewitt

Computes the edge strength components using

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

6.1.2 Sobel

Computes the edge strength components using

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

This operator provides greater resilience to noise and is the best estimator of edge orientation and strength of all the "small" kernels.

6.1.3 Canny

Canny took an information theoretic approach to edge detection, stating that an edge detector should

1. Detect an edge
2. Should give a response in the correct location
3. Have a single response to an edge

He assumed that he was searching for an ideal step edge in the presence of Gaussian noise and defined a matched filter that could be approximated by the difference of a Gaussian. Although this gives the optimal edge detector for images with only this type of noise corruption, it responds adequately for other noise distributions.

The operator first convolves the image with a Gaussian kernel to perform the noise reduction (just as the Prewitt operator has the regions of 1s to perform averaging). It then differentiates the image in the two orientations. Rather than perform one convolution for smoothing and a further two for differentiation, the smoothing and differentiation kernels are combined and the whole operation is performed using two convolutions. (this works because convolution is associative: $\nabla \bullet (G \bullet I) = (\nabla \bullet G) \bullet I$.)

The amount of noise reduction may be controlled by varying the standard deviation, σ, in the Gaussian function. Larger values of σ imply more smoothing: more noise reduction and also more blurring of the edge information.

6.1.4 Marr-Hildreth

The major disadvantage of using differentiation for edge detection is in locating the edge accurately. This problem is avoided by double differentiating the data. Consider the cross section through an ideal step edge that has been blurred. Differentiating once will result in a spike output whose width is proportional to the degree of blurring. Single differential operators discussed so far will have to locate the maximum in this cross section. But if it is differentiated again we will obtain a profile that is zero in the uniform region either side of the edge and has positive and negative going excursions either side of the edge: the profile crosses the zero axis at the edge. This is the principle of the double differentiation operators.

The Marr-Hildreth is currently the most favoured of these operators. It uses Gaussian smoothing to reduce noise followed by convolution with the Laplacian operator. Again, the Laplacian can be convolved with the Gaussian beforehand and the LoG convolved with the image.

By varying the value of σ, we can control the size of the smoothing element and hence control the size of the objects that are selected by the filter.

6.1.5 MATLAB Code

MATLAB's Image Processing Toolbox provides a generic edge detection function

```
>> BW = edge(src, 'sobel', thresh, direction);
>> BW = edge(src, 'prewitt', thresh, direction);
>> BW = edge(src, 'log', thresh, sigma);
>> BW = edge(src, 'canny', thresh, sigma);
```

The log filter is an alternative name for the Marr-Hildreth. The image returned is the same size as the source, but binary: a value of 1 indicates that a pixel is flagged as being on an edge. The source image must be greyscale. In all cases, the third and fourth arguments are optional.

For the Prewitt and Sobel detectors, the optional `thresh` argument can be used to specify an edge strength threshold, if it is omitted, the threshold is computed automatically. `direction` is used to specify which orientation of edge detector is to be used, the default is to use both. There are other arguments, but they should not be needed.

For the Marr-Hildreth detector (log), `thresh` is an edge strength threshold, `sigma` defines the width of the smoothing function, its default value is 2.

Canny requires two threshold values, so `thresh` can be a vector, or a scalar in which case the two thresholds will be `thresh` and 0.4*`thresh`. Again, `sigma` defines the width of the smoothing function, but the default value is 1.

6.2 Low Pass – Smoothing

The aim of smoothing algorithms is to reduce the small scale, small amplitude fluctuations in the data, usually associated with noise. (Noise is due to a multitude of causes, too many to list, it is observed in one of two ways: salt and pepper noise where isolated individual pixels have *very* different values to their neighbours', and random noise where all pixels' values may differ from the correct value – the statistics of the differences can be described.)

6.2.1 Averaging

The simplest approach to reducing noise is to replace each pixel's value with the average computed over some neighbourhood around it. The noise amplitude is reduced by a factor equal to the square root of the number of pixels involved in the smoothing. Any shape neighbourhood can be used, but for simplicity of computation, square regions are usually chosen, having an odd number of pixels on the side.

Whilst the algorithms are simple, they do have the disadvantage of smoothing everything in the image, this includes image detail that we might want to preserve. A solution to this is to compute the smoothed value and compare it with the value that is to be replaced. Only if the two are reasonably similar is the smoothing is performed. If there is a significant difference we assume that the neighbourhood spans a significant edge in the image: the central pixel is on one side of the edge, there are sufficient pixels in the neighbourhood from the other side of the edge to influence the average.

Averaging is achieved by convolving the image with a simple template:

$$\begin{bmatrix} \frac{1}{n} & \cdots & \frac{1}{n} \\ \vdots & \ddots & \vdots \\ \frac{1}{n} & \cdots & \frac{1}{n} \end{bmatrix}$$

where *n* is the number of template elements.

6.2.2 Weighted Averaging

Weighted averaging is a generalisation of the simple averaging. Instead of giving equal weight to all pixels in the neighbourhood, we vary the contribution made by each pixel, commonly giving larger weight to the pixels closest to the centre of the neighbourhood. The smoothed image will resemble the original more closely than was the case with simple averaging.

6.2.3 Gaussian Smoothing

Gaussian smoothing is a special case of weighted smoothing, where the coefficients of the smoothing kernel are derived from a Gaussian distribution. The amount of smoothing can be controlled by varying the values of the two standard deviations (the distribution has a σ value for the x and y orientations). We can therefore control the size and, to a certain extent, the shape of the feature to be smoothed. Gaussian smoothing has several mathematical advantages, the most important of which is that the effects of using a large kernel can be approximated by repeated use of smaller kernels.

6.2.4 Rank-Order Filters

The one significant disadvantage to the smoothing operators discussed above is the difficulty they have in differentiating between features that should be retained from noise that should be removed. Rank order filters can do this, at the expense of significantly greater amounts of processing.

Consider the example of a profile through a real edge (a step edge that is blurred and noise corrupted). The aim of smoothing is to reduce the noise fluctuations but retain the underlying structure. Simple smoothing will reduce the noise but also blur the structure. Rank order filters can reduce noise and retain structure.

A rank order filter will take the set of pixels within some neighbourhood, rank them in order of increasing intensity and select, for example, the one value in the middle of the list (if the list has an odd length, this is simple, if the list length is even, the average of the two central values would be chosen). The filter that chooses the central value is known as the median filter. Naturally, different effects can be achieved by choosing other values.

MATLAB provides tools for performing rank-order filtering (in the past it was known as order-statistic filtering):

```
>> dst = ordfilt2(src, order, domain);
```

The filter gives the order-th element in the neighbourhood specified by domain. (And domain can be specified by the MATLAB function to generate a matrix filled with ones: ones(m, n).) So the median filter could be realised by (with m and n suitably intitialised):

```
>> dst = ordfilt2(src, m*n/2, ones(m, n));
```

Or by a call to the specific median filter:

```
>> dst = medfilt2(src, [m, n], padopt);
```

The optional argument padopt specifies how the border pixels are treated: 'zeros' (the default) pads the border with zeroes, 'symmetric' pads the border with a mirror image of the border pixels, and 'index' pads the border with 1s if the image is of class double, or 0s otherwise.

6.3 Morphology

Morphology is the study of the form of things. Morphological transforms are designed to find this structure. Morphological transforms are usually applied to thresholded data, but can equally well be defined for greyscale and colour images. There are two fundamental transforms: erosion and dilation that can be combined in two ways to give two derived operations: opening and closing. Further combinations are defined to achieve other effects.

Morphological operations firstly require a structuring element. This is simply a kernel that defines a shape, commonly a circle, square or cross, others are equally possible and may be more useful in specific circumstances. The values in the element are unimportant. The element has an origin that specifies its location in the image and therefore the position where the operation's result will be written.

6.3.1 Dilation and Erosion

These are most simply defined for binary images (pixels have values of 1 or 0 only). A structuring element is said to fit at a location in a binary image if *all* the image pixels that overlap the structuring element have value 1. A structuring element is said to hit at an image location if *any* of the pixels overlapping the structuring element are 1.

Binary erosion of an image *f(x,y)* by a structuring element *s(x,y)* is defined by

$$f \otimes s = \begin{cases} 1 \text{ if } s \text{ fits } f \\ 0 \text{ otherwise} \end{cases}$$

The effect of erosion is to shrink objects: pixels are removed from the boundary of the object to a depth approximating half of the structuring element's width. Objects of this characteristic size are removed completely.

Binary dilation is similarly defined as

$$f \oplus s = \begin{cases} 1 \text{ if } s \text{ hits } f \\ 0 \text{ otherwise} \end{cases}$$

Its effect is to increase the size of an object by adding pixels to a depth of about half the structuring element's width. Gaps of this size in the object are filled.

The definitions can be modified for greyscale images. Rather than setting the output to zero or one, the minimum or maximum of the set of values selected by the structuring element are selected. So greyscale erosion is defined as

$$f \otimes s = \min_{j,k \in s} \{ f(m-j, n-k) - s(j,k) \}$$

And greyscale dilation as

$$f \oplus s = \max_{j,k \in s} \{ f(m-j, n-k) - s(j,k) \}$$

Structuring elements differ according to whether the image to be manipulated is binary (a flat se) or greyscale (a non-flat se). The basic syntax for creating a structuring element is:

```
>> se = strel(shape, parameters);
```

A number of predefined shapes are available:

Shape	**Parameters**	
diamond	R	distance from origin to extreme points of the diamond
disk	R	radius (plus other parameters)
line	length, orientation	length of element and angle wrt horizontal axis
octagon	R	length from origin to a vertical side, must be a multiple of 3
pair	offset	is a vector specifying an offset
periodicline	P, V	a se of 2P + 1 elements, each separated by the vector V. Element 1 is at the origin, element 2 at -1*V, etc.
rectangle	MN	a two element vector of non-negative values: the numbers of row and columns in th ese
square	W	the dimension of the square
arbitrary	NHOOD	creates an arbitrary shaped se, NHOOD is a matrix of 0s and 1s specifying the shape

Non-flat structuring elements are created by passing strel two matrices, the first defines the neighbourhood, the second the height at each point in the neighbourhood.

In MATLAB, erosion and dilation of both binary and greyscale images are achieved by:

```
>> dst = imerode(src, se);
>> dst = imdilate(src, se);
```

6.3.2 Opening and Closing

Opening is defined to be erosion applied a number of times, followed by dilation applied the same number of times with the same structuring element. Noise structures are removed and the features shrunk by erosion, the features are then restored by dilation.

Closing is defined as dilation followed by erosion.

If we apply both opening and closing operations to a greyscale image, we achieve a degree of smoothing.

Opening and closing are invoked by:

```
>> dst = imopen(src, se);
>> dst = imclose(src, se);
```

(open(src) and close(src) can be used and they assume a 3x3 se.)

6.3.3 Top Hat

The top hat operator is defined as the difference between an image and the opened version. It enhances detail that would otherwise be hidden in low contrast regions. It is invoked by imtophat if a structuring element is to be specified or tophat if a 3x3 element is to be assumed.

6.3.4 Edge Detector

The morphological edge detector is defined as the difference between the opened and closed images.

6.3.5 Skeleton

The skeleton of a binary object is a one-pixel thick line that represents the object's shape. The skeleton retains shape and topology but loses thickness information. Obvious uses are in applications where these factors are important, such as character or fingerprint recognition. MATLAB provides the function `skel`.

7 Image Processing

Image processing operators compute the value of a single pixel in the output using all of the input values.

7.1 Fourier Transform

The Fourier transform was suggested in 1807. It states that any periodic function can be synthesised as the sum of sines and/or cosines of different amplitudes and frequencies: the set of amplitudes is known as a Fourier series. Even non-periodic functions can be represented by these so-called basis functions (the sines and cosines). The amplitudes of the sines and cosines are computed using a Fourier transform.

The original data can be recovered exactly using an inverse Fourier transform. So it is possible to transform the data to the "Fourier domain", process it and inverse transform back to the original domain.

The important point to understand in the Fourier transform relates to spatial frequencies. A low spatial frequency corresponds to a large object – think of a slow wave. Conversely, a high spatial frequency corresponds to rapid fluctuations. Or we can view this from the viewpoint of image structures: the smaller an image structure, the broader the range of spatial frequency components that is required to synthesis it.

Mathematically, the Fourier transformed data is in two parts: the magnitude and phase. The magnitude part tells us how much energy there is at a given spatial frequency, the phase tells us the relative starting point of each component. Both parts are needed for the reverse transform, but we are interested in manipulating only the magnitude.

If we compute the Fourier transform of an image:

```
>> dst = fft2(src);
```

The magnitude of the transform is returned by

```
>> dst = abs(src);
```

and scale the output linearly to make it more visible

```
>> imshow(src, [ ]);
```

We will observe the zero frequency component in the four corners of the image (this corresponds mathematically to the mean grey value). The transform can be simplified by moving the origin to the centre:

```
>> dst = fftshift(src);
```

Frequencies increase away from the centre.

We see that the transform data is symmetrical: $F(u,v) = F(-u,-v)$. An image with many small objects will have more high frequency content. Scaling non-linearly can make the faint data easier to see:

```
 >> dst = log(1+ src);
>> imshow(dst, [ ]);
```

Where `src` in this case is the Fourier transformed data.

The reverse of these transforms is as follows. `ifftshift(src)` rearranges the quadrants of the data, `ifft2(src)` performs the inverse Fourier transform.

The Fourier transform that is implemented works most efficiently if the image dimensions are powers of 2.

7.1.1 Convolution Theorem

It can be shown that the convolution of two functions will give the same result as multiplying the functions' Fourier transforms. Although it would seem that there is more computation to be performed if we Fourier transform the image, multiply it by some filter and compute the inverse, there is actually more computation to be performed in the direct convolution if the kernel is sufficiently large; somewhere around 15x15 pixels.

Therefore, if we can identify the Fourier transform of the earlier kernels, we can achieve the results of the convolution in the frequency domain.

A certain type of error is avoided if the two images to be processed are padded with zeros. If the original image dimensions are *(A, B)* and *(C, D)*, the padded images, usually the same size, must be at least *(A + C – 1, B + D – 1)*. To make the transforms more efficient, the padding is usually such that the image dimensions are now powers of 2.

7.1.2 High-, Low- and Band-Pass filters

Edge detection is often thought of as enhancing small-scale structures: edges. It should not be a surprise therefore that high pass filtering is equivalent to edge detection.

The simplistic high pass filter is defined as:

$$H(u,v) = \begin{cases} 1 \text{ if } D(u,v) \geq D_0 \\ 0 \text{ otherwise} \end{cases}$$

Where *D(u,v)* computes the distance of the point *(u,v)* from the origin.

Whilst this is the obvious filter, and is sometimes called the *ideal* high-pass filter, it introduces serious artefacts that mean it should never be used. Instead, a high-pass filter with a more gradual cutoff should be used.

The ideal low-pass filter is defined similarly, and should also not be used. A Gaussian function could be used instead. The maximum value of the filter should be 1. A high-pass filter can be made from this low-pass one by subtracting it from 1.

A band-pass filter is a combination of high- and low-pass filters, with the high-pass filter's cutoff frequency less being than the low-pass filter's one.

7.1.3 Wiener Filter

The Wiener filter is designed to remove image degradations, provided we are able to model the degradation. Degradations such as relative motion of the camera and subject, out-of-focus optics or light scattering can be modelled. We assume that the observed image, *f*, is generated by adding noise, *n*, to the ideal image, *g*, that is convolved with some function, *H*, that represents the total imaging system (i.e. includes the degradation).so the observed image can be described as:

$$f = gH + n$$

The process of computing *g* from *f*, assuming *n* and *H*, is called deconvolution. MATLAB provides a single function to perform it:

```
>> dst = deconvwnr(src, PSF);
```

PSF is the degradation model that you assume. The function has further arguments that can improve the results – if their values are known accurately.

Appendix A - Matrix Manipulation

Remember that a matrix is an n x m array of values. An individual element is referenced as **M**[r,c]. All the values are of the same type.

A.1 Addition and Subtraction

These are defined iff the numbers of rows and columns are the same. Then they are defined as elementwise addition (or subtraction):

$$\mathbf{R}[\mathbf{r},\mathbf{c}] = \mathbf{A}[\mathbf{r},\mathbf{c}] + \mathbf{B}[\mathbf{r},\mathbf{c}]$$

A.2 Multiplication

Is defined only if the number of columns of matrix 1 equals the number of rows of matrix 2. Then each element of the result is computed by:

$$\mathbf{R}[\mathbf{i},\mathbf{k}] = \sum_{\mathbf{j}} \mathbf{A}[\mathbf{i},\mathbf{j}] * \mathbf{B}[\mathbf{j},\mathbf{k}]$$

The order of multiplication is important, as it will affect the result. Generally **AB** ≠ **BA**.

A.3 Identity

The identity matrix is defined as a square matrix with the elements on the leading diagonal (the same row and column indices) having the value 1, every other element is zero. It is given the symbol **I**.

Multiplying any matrix by the identity leaves the matrix unchanged.

A.4 Inverse

The inverse of a matrix is denoted as $\mathbf{M}^{-1}$. The product of a matrix and its inverse is defined to be the identity. This is an exception to the order of multiplication rule, in that $\mathbf{M}\,\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M}$.

A.5 Transpose

The transpose of a matrix is given by interchanging the rows and columns. So what was an n by m matrix becomes m by n, and the elements are interchanged: **M**[r,c] = **M**[c,r]. The transpose of a matrix is denoted by $\mathbf{M}^{T}$.

Appendix B – Vector Processing

A vector has magnitude and direction. It can be resolved into components, i.e. a set of vectors that add up to the original. Whilst a vector can be resolved into any number of components in many orientations, it is most usual to resolve into two components that are at right angles. The benefit here is that each component is completely independent of the other.

The reverse problem happens when making measurements, e.g. of edge strength. We have the two components (edge strengths parallel to the image sides), and we want to find the vector strength and magnitude. This is done using the triangle of vectors: the two components are drawn head to tail to form a right angled triangle. The resultant vector is the triangle's hypotenuse. Its magnitude is found as the length of the hypotenuse, using Pythagoras, its orientation is the inverse tangent of the ratio of the two components:

$$|\mathbf{V}| = \sqrt{\mathbf{V}_x^2 + \mathbf{V}_y^2}$$

$$\theta = \tan^{-1} \frac{\mathbf{V}_y}{\mathbf{V}_x}$$

Appendix C - MATLAB Programming

MATLAB allows sequences of instructions to be stored in so-called M-files.

An M-file has the following structure:

- The function definition line
- The H1 line
- Help text
- The function body
- Comments

The function definition line looks like

```
function [outputs] = function_name(inputs)
```

For example, a function to split a colour image into its R, G, and B components would look like:

```
function [R, G, B] = split_colour_planes(src)
```

This would be called at the command prompt as:

```
>> [r, g, b] = split_colour_planes(a_colour_image);
```

There are restrictions on the form of function names: they must begin with a letter, can't contain any spaces and only the first 63 characters are significant. If the function has no output, the square brackets and equal sign are omitted. If the function has a single output, the square brackets may be omitted.

The H1 line follows the definition line with no blank lines of leading spaces. It is a comment that Matlab returns first when `help` is invoked, and is searched for keywords when `lookfor keyword` is used. The format of the H1 line is

```
% function_name <text description>
```

The help text must follow with no blank lines. The lines up to the next blank line are also printed by `help`. Each line of help text must start with a % symbol.

Other lines stating with the % symbol are treated simply as comments for the programmer.

The function body can contain any instruction that would be valid at the command line. Array and matrix operations are shown in Table 1, image arithmetic operations in Table 2.

Matlab provides the usual programming constructs, but (naturally) the syntax is different to other languages.

Conditional execution

```
if expression
    statement(s)
end
```

OR

```
if expression1
    statement1
elseif expression2
    statement2
else
    statement3
end
```

For loop

```
for index = start:increment:end
    statements
end
```

While loop

```
while expression
      statements
end
```

break   Terminate loop execution

continue   Passes control to the next iteration of a loop

Switch: executes one of several alternative statements

```
switch  switch_expression
    case case_expression1
          statement(s)
      case  case_expression2
          statement
      otherwise
          statement
end
```

return   Causes execution to return to the calling function

try … catch   changes flow of control if an error is encountered

Operator	**Name**	**Matlab function**	**Examples**
+	Addition	plus(A, B)	a + b
-	Subtraction	minus(A,B)	a - b
.*	Array multiplication	times(A,B)	C = A.*B, C(I,J)=A(I,J)*B(I,J)
*	Matrix multiplication	mtimes(A,B)	A*B, standard matrix multiplication
./	Array right division	rdivide(A,B)	C = A./B, C(I,J) = A(I,J)/B(I,J)
.\	Array left division	ldivide(A,B)	C = A.\B, C(I,J) = B(I,J)/A(I,J)
/	Matrix right division	mrdivide(A,B)	A/B, is roughly A*inv(B)
\	Matrix left division	mldivide(A,B)	A\B, is roughly inv(A)*B
.^	array power	power(A,B)	C = A.^B, C(I,J) = A(I,J)^B(I,J)

^	Matrix power	mpower(A,B)	C = mpower(A,x) is $A(I,J)^x$
			C = mpower(x,A) is C(I,J) = $x^{A(I,J)}$
			C = mpower(A,B) is an error

.’	Vector, matrix transpose	transpose(A)	A.’ as standard
‘	Vector, matrix complex conjugate	ctranspose(A)	A’ as standard
+	unary plus	uplus(A)	+A
-	Unary minus	uminus(A)	-A
:	Colon		Discussed in section 4.2

Table 1: Array and Matrix Operations

Function	**Description**
imadd	Adds two images, or an image and constant
imsubtract	Subtracts two images, or a constant from an image
immultiply	Multiplies an image by a constant, or two images pixel by pixel
imdivide	Divides an image by a constant, or two images pixel by pixel
imabsdiff	Computes absolute difference between two images
imcomplement	Computes the negative
imlincomb	Computes a linear combination, A*a + B*b + C*c + …, A is an image, a a scalar

Table 2: Some Image Processing Toolbox Operators

References

R.C. Gonzalez, R.E. Woods, S.L. Eddins, (2004) *Digital Image Processing Using Matlab*, Prentice Hall, Upper Saddle River, New Jersey.