AMME4701 Computer Vision and Image Processing
Tutorial Activity Week 1, 2022

THE UNIVERSITY OF SYDNEY

Introduction to the MATLAB Image Processing Toolbox

- This activity is to be completed during the Week 1 tutorial (Friday 5th August 2021)
- When you have completed the activity, you will need to show your results to the tutor who will check your work and that you have sufficiently completed the task: this needs to be completed during the tutorial time, so make sure you are ready to show the tutors your work during the last hour of the tutorial.

Objectives

- This tutorial will introduce you to basic techniques in image processing, the use of the MATLAB Image Processing Toolbox, and get you thinking about some of the issues in processing, interpreting and analysing image data.
- You should step through and experiment with each of the functions described using the example images in 'example_images_week1.zip'. Use `help <function_name>` to get more details on how to use each function. Try experimenting with the additional options in each function.
- You should complete each of the activities highlighted in the boxes throughout the sheet. Feel free to ask the tutors for help if you run into problems.

1. Background: Digital Images and the MATLAB Image Processing Toolbox

MATLAB provides a convenient set of functions and apps for image processing, analysis, visualisation and algorithm development through the Image Processing Toolbox (https://au.mathworks.com/products/image.html). Mathworks provides many examples for the use of these tools that you can explore to get a feeling for the potential use of these tools through their website.

1.1 Reading images

Images are read into the MATLAB environment using the function `imread`, where the first argument to the function is a string containing the complete file name of the image file (including extension). For example:

```
f = imread('apples.jpg');
```

reads the JPEG image myimage.jpg into an array f. The function `imfinfo` provides some details on the image without loading the image data contained in the file:

```
K = imfinfo('apples.jpg');
```

Where information about the image 'myimage.jpg' is stored in a struct K (for example, data such as the image width and height are stored in the fields K.Width and K.Height).

1.2 Image representation: Image data in MATLAB is stored as a multi-dimensional array of size [M x N x D] with M rows and N columns representing the image data per pixel of an image of width N and height M (in pixels). D corresponds to the number of channels in the image data (e.g. 1 for greyscale and binary images, 3 for Red-Green-Blue (RGB) colour images). For RGB-colour images, the third array index represents colour channel: 1: red, 2: green, 3: blue. You can use the function `size` to see the row and columns dimensions of the image (the array): i.e. `size(f)` or `[M,N,D] = size(f)`. The `whos` function returns information about the array, in particular its size and data class.

Each element of the image data array contains information about the corresponding value or brightness of that pixel. The following table lists the various data classes supported by MATLAB and the Image Processing Toolbox for representing pixel values. The first eight are referred to as numeric data classes:

Name	Description
`double`	Double-precision, floating-point numbers, range -10^{308} to 10^{308} (8 bytes per element)
`uint8`	Unsigned 8-bit integers in the range [0,255] (1 byte per element)
`uint16`	Unsigned 16-bit integers in the range [0,65535] (2 bytes per element)
`uint32`	Unsigned 32-bit integers in the range [0,4294967295] (4 bytes per element)
`int8`	Signed 8-bit integers in the range [-128,127] (1 byte per element)
`int16`	Signed 16-bit integers in the range [-32768,32767] (2 bytes per element)
`int32`	Signed 32-bit integers in the range [-2147483648,2147483647] (4 bytes per element)
`single`	Single-precision, floating-point numbers, range -10^{38} to 10^{38} (4 bytes per element)
`char`	Characters (2 bytes per element)
`logical`	Values are 0 or 1 (1 byte per element)

Common image formats typically use 8 bits per channel (i.e. 0 for completely dark and 255 for completely bright), although some applications call for images data at a greater 'bit depth', or data types which are non-integer (i.e. floating point images). Logical/binary types may be used for images representing masks. Image data can cropped, transformed and manipulated using standard MATLAB array operations:

```
im = imread('apples.jpg');
imr = im(:,:,1); % get red channel
img = im(:,:,2); % get green channel
imb = im(:,:,3); % get blue channel

% crop 50 x 50 pixel region starting at (100,120)
im_crop = im(100:150,120:170,:);

% swap the red and blue image channels
im2 = cat(3,im(:,:,3),im(:,:,2),im(:,:,1));

% sub-sample every 2nd pixel in vertical direction and
% 3rd pixel in horizontal direction
im_subsamp = im(1:2:end,1:3:end,:);

im_flipped = im(end:-1:1,:,:); % flip image (vertical)
```

1.3 Displaying Images: Images can be displayed in MATLAB using the function `imshow(f)` where f is an image array. You can specify a range of intensities by typing:

```
imshow(f, [low high])
```

In that case, all values in the image which are less than or equal to low are displayed as black, and all values greater than or equal to high are displayed as white. Values in between are displayed as intermediate intensity values using the default number of levels. You can also 'expand' the dynamic range of an image automatically by:

```
imshow(f, [ ])
```

which sets the previous variable low to the minimum value of the array f and high to its maximum. This is useful for displaying images with a low dynamic range.

When an image is already open in a figure, the functions `impixelinfo` and `imdistline` provide the ability to display the intensity values (or RGB values for a colour image) of individual pixels or measure the distance between pixels by using the cursor, for example:

```
f = imread('apples.jpg');
impixelinfo;
imdistline;
```

1.4 Writing Images: Images are written to disk using the function `imwrite`:

```
imwrite(f, filename)
```

where *filename* includes a recognised file format extension (check the accepted formats). You can also specify the desired format explicitly using a separate third argument:

```
imwrite(f, basename, extension)
```

The `imwrite` function can have other parameters, depending on the file format specified. For example, to save an image in the JPEG format, you can specify a quality parameter which will set the level of compression achieved:

```
imwrite(f, 'filename.jpg', 'quality', q)
```

where q is an integer between 0 and 100 (the lower the number, the higher the degradation due to JPEG compression, but the smaller the size of the file).

1.5 Converting between Data Classes: To convert between data classes one can do: B = data_class_name(A) where data class name is one of the names of the available data classes (see the above table). When converting between data classes in this way, it is important to keep in mind the value ranges for each data class. The toolbox provides specific functions that perform the scaling necessary to convert between images classes and types:

Function:	Convert input to:	Valid input data classes:
im2uint8	uint8	logical,uint8,uint16,double
im2uint16	uint16	logical,uint8,uint16,double
mat2gray	double (in range [0, 1])	double
im2double	double	logical,uint8,uint16,double
im2bw	logical	uint8,uint16,double

1.6 Arithmetic operations on image data: Several applications call for applying arithmetic operations on images (such as scaling pixels by a constant (multiply or divide), finding a weighted combination of image values (addition and multiplication) or finding the difference between two images). When performing arithmetic operations on image data one must take care to observe that integer types have minimum and maximum values that might be invalidated by certain operations.

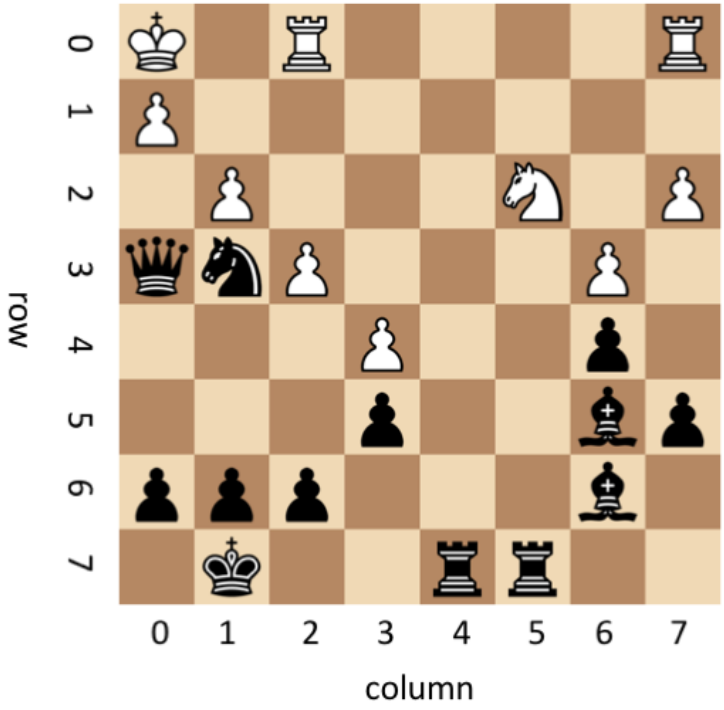
It is possible to apply for loops in MATLAB to increment across the elements of an image array (in two dimensions or more) are perform these operations pixel-by-pixel. However, this approach does not work well for large image arrays, as MATLAB for loops are much slower than in other languages, such as C and C++. MATLAB provides functions for image arithmetic that are much more efficient than using loops, and automatically account for min/max values for most data types, and these should be used where possible:

MATLAB Image Arithmetic Functions

imabsdiff	Absolute difference of two images
imadd	Add two images or add constant to image
imapplymatrix	Linear combination of color channels
imcomplement	Complement image
imdivide	Divide one image into another or divide image by constant
imlincomb	Linear combination of images
immultiply	Multiply two images or multiply image by constant
imsubtract	Subtract one image from another or subtract constant from image

You should test out each of these functions with one or more of the example images provided to get a feeling for exactly how they work.

Tutorial Activity 1: Basic Image Operations



Open the image ‘chess.png’, which contains a visual depiction of a chess game.

- Write a MATLAB function that provided with two arguments (row, col) which correspond to a board position (see above figure). The function should produce and display a cropped image of the corresponding board position, displaying the piece that currently occupies this position.
- Try extending the function such that it indicates if the current square is empty, contains and white piece or contains a black piece. Hint: white pieces are mostly made of exactly white pixels (i.e. RGB = [255,255,255]) and black pieces mostly made of exactly black pixels (i.e. RGB = [0,0,0]) (if you have trouble, try moving onto the next activity and coming back to it)

1.7 Image histograms: It is often useful to be able to visualise a histogram of the values present in an image. MATLAB provides the function `imhist(f)` to plot a histogram for a single channel, where f in this case must be an M x N x 1 array. For example:

```
im_gray = imlincomb(0.33,im(:,:,1), 0.33,im(:,:,2), 0.33,im(:,:,3));
imhist(im_gray)
```

Converts a colour image to grayscale and plots a histogram of the grayscale intensity values (see also `rgb2gray`). Another example:

```
im = imread('apples.jpg');
figure;
subplot(3,1,1)
imhist(im(:,:,1)) % red channel
ylabel('red')
subplot(3,1,2)
imhist(im(:,:,2)) % green channel
ylabel('green')
subplot(3,1,3)
imhist(im(:,:,3)) % blue channel
ylabel('blue')
```

will display a figure with three subfigures, each containing a histogram of each of red, green and blue channels separately.

1.8 Histogram Equalisation: During the week 1 lecture, we discussed algorithms for contrast stretching and histogram equalisation. MATLAB provides a few functions to perform these operations. The function `imadjust`:

```
im_adjusted = imadjust(im, [LOW_IN; HIGH_IN],[LOW_OUT; HIGH_OUT]);
```

can be used to apply an offset and scaling to the image array 'im' that maps values from LOW_IN to LOW_OUT and HIGH_IN to HIGH_OUT, where values in-between are mapped in a linear fashion between these two points. For example:

```
im_adjusted = imadjust(im, [0; 0.1], [0; 1]);
```

would produce a new image array of the same type as 'im', but where values in the original image between 0 and 0.1 have been spread out to the entire dynamic range [0, 1] in the output image. Values in the original image greater than 0.1 are clipped to a value of 1 in the new image.

The function `histeq` performs histogram equalisation on image data, returning image data that has been transformed through a monotonically increasing, non-linear function for which the resulting image data has a roughly uniform histogram. MATLAB also provides a spatially-adaptive histogram equalisation function `adapthisteq`, which provides options for varying the degree of contrast limitation and the resolution of the spatial blocks over which the histograms are evaluated and applied. Try loading up the example image 'pagetext.png' and applying both histogram equalisation and adaptive histogram equalisation to this image to make the text more readable on both sides of the page

Tutorial Activity 2: Histograms and Contrast Adjustment

Open up the example images ‘fish001.jpg’, ‘fish002.png’ and ‘underwater001.png’. Images taken underwater are typically dark, exhibit poor contrast and have a blue-green colour cast (due to attenuation of red light in water).

- Use linear contrast enhancement (e.g. `imadjust`), histogram equalisation and adaptive histogram equalisation methods (e.g. `histeq` and `adapthisteq`) to improve the quality of the images.
- Try applying these methods to a grayscale version of the image and to each of the colour channels separately. Have a play with the different parameters of each method and see how many fish you can observe clearly.

1.9 Image thresholding: Thresholding is the process of assigning pixels in an image to discrete classes based on whether the pixel's value is above or below a given threshold. Thresholding typically applies to two classes (i.e. true/false, on/off) but potentially may involve being assigned to one of many classes (i.e. levels). Thresholding is useful for various tasks in computer vision, for example the removal of foreground/background objects.

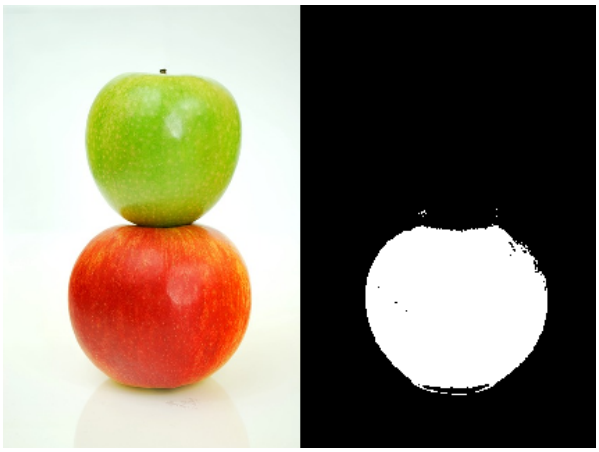
To produce a binary image based on a threshold image value V, one can use operations >, >=, <, <= on image arrays. For example, where 'im_gray' is a grayscale image array with one channel (i.e. size(im_gray,3) = 1:

```
im_bin = im_gray < 120;
```

produces a binary image with pixel values 1/True where the corresponding pixel in im_gray is less than 120 and pixel values 0/False for those that are equal or greater than 120. When these operations are applied to images with multiple channels, the output is a multi-dimensional image of the same dimensions, where each channel of the output corresponds to the binary image for each image channel (i.e. colour intensities thresholded separately, channel by channel). The logical operations AND/OR/NOT can additionally be applied to provide a final decision based on multiple thresholds. In MATLAB you can either use the functions `and`, `or`, `not` or the operators `&` `|` `~` (note the use of single characters for element-by-element operations). For example:

```
im2 = im(:,:,1) > 180 & im(:,:,2) < 170 & im(:,:,3) < 128;
```

produces a binary image where values are true only if the red channel value is greater than 180 and the green channel is less than 170 and blue channel less than 128, which roughly segments the red apple:



MATLAB provides implementations of algorithms for automatically assigning thresholds (for example `graythresh`, `imbinarize` and `otsuthresh` which use Otsu's method for thresholding, see week 1 lecture). `imbinarize` also provides an option for performing locally-adaptive thresholding (by using the `'adaptive'` argument) that computes a different threshold value for each pixel based on the histogram of the local surrounding area. Try experimenting with `imbinarize` on the example image 'pagetext.png' using both the `'global'` and `'adaptive'` methods.

Tutorial Activity 3: Image Thresholding

Otsu's method can also be extended to multiple classes (greater than two) with separate thresholds (for example find the best two thresholds to segment an image into three classes). MATLAB provides an implementation of this method in the functions `multithresh` and `imquantize`. Try experimenting with these functions on the example image 'apples.jpg'.

- Load the colour image 'apples.jpg' and convert it to grayscale using the function `rgb2gray`.
- Develop code to segment the image into three classes: background, red apple, green apple using `multithresh` and `imquantize`.
- Plot the segmented image side-by-side with the original. For the segmented image, produce a graph were red apple pixels are coloured red, green apples pixels green and background pixels white.
- Produce a plot of the grayscale image histogram and draw vertical lines on top of the histogram indicating the positions of the thresholds

Note where the algorithm has correctly and incorrectly segmented the image.

Another more effective method for segmenting this image is based on colour. We will explore this during next week's lecture and tutorial.

1.10 Some important standard arrays: The following MATLAB functions can be useful to generate simple image arrays to try out ideas, test the syntax of functions during development, or test some image processing algorithms:

- `zeros(M,N)` generates an M × N matrix of 0s of class double
- `ones(M,N)` generates an M × N matrix of 1s of class double
- `true(M,N)` generates an M × N logical matrix of 1s
- `false(M,N)` generates an M × N logical matrix of 0s
- `magic(M)` generates an M × M "magic square": a square array in which the sum along any row, column or main diagonal, is the same (numbers are integers).
- `rand(M,N)` generates an M × N matrix whose entries are uniformly distributed random numbers in the interval [0, 1]
- `randn(M,N)` generates an M × N matrix whose entries are normally distributed (i.e. Gaussian) random numbers with mean 0 and variance 1
- `checkerboard(N,P,Q)` generates a checkerboard pattern of P x Q tiles, each tile N x N in size.

If only one argument is included in any of the previous functions, the result will be a square array.

1.11 Other useful functions: You should investigate the use of (or at least be aware of) these functions:

`rgb2gray`	Convert an RGB colour image or colourmap to grayscale
`cat`	Concatenates arrays along a specified dimension (useful, for example, to combine multiple channels into a single colour image)
`imshowpair`	Compare images side-by-side
`montage`	Display multiple image frames as rectangular montage
`label2rgb`	Converts a labelled image into a colour image, different colours per label
`imresize`	Resize an image using interpolation

Tutorial Activity 4: Spot the Difference (optional)

- Load the image 'spot_the_difference.png'. It contains two images side by side that are identical except for a few artistic changes. Spot the differences!
- Actually, better yet, write a script that produces a helper/solution image to help someone spot the differences, like this:

- If you are really keen, try to write an algorithm to count the differences. Have a look at the MATLAB function `bwconncomp`. It computes a cell array corresponding to the connected components of an image (i.e. regions for which all adjacent pixels are of the same value). The algorithm cycles through each unexplored pixel and performs a "flood-fill", until all pixels are labelled. See http://au.mathworks.com/help/images/ref/bwconncomp.html or `help bwconncomp` for more details. We will explore these types of functions in the next few weeks.

References and Further Reading:

[1] R.C. Gonzalez, R.E. Woods and S.L. Eddins, Digital Image Processing using MATLAB, Prentice Hall, 2008.

[2] MATLAB Image Processing Toolbox Documentation, https://au.mathworks.com/products/image.html, Mathworks, 2022