**PRACTICAL**

**Introduction to Image Analysis**

**Wednesday, 15th January 2025**

**Maria Anastasiadi (**[**m.anastasiadi@cranfield.ac.uk**](mailto:m.anastasiadi@cranfield.ac.uk)**)**

# Objectives

In metabolomics, computational methods are often used to interpret visual data. This can be as simple as measuring the colour intensity of a time-series of images, or as complex as a convolutional neural network for object identification purposes. **This practical aims to introduce you to the basics of computer vision techniques, from basic adjustments to object segmentation**. We will mainly make use of the package **EBImage** from Bioconductor, which provides general purpose functionality for image processing and analysis.

EBImage is particularly useful for microscopy-based cellular assays, as it offers tools to segment cells and extract quantitative cellular descriptors. During this practical we will study two-dimensional cell images, and a strawberry plant image, and learn how to identify the objects positions and shapes and how to quantitatively measure characteristics of the identified shapes and patterns, such as sizes, intensities, colour distributions and relative positions.

To install EBImage you will need R version "4.4.2". Type:

if (!require("BiocManager", quietly = TRUE))

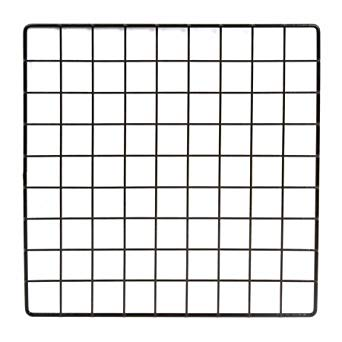
install.packages("BiocManager")

BiocManager::install("EBImage")

You will also need to load the package to be able to use it during the practical session:

# library("EBImage")

# Intro to Computer Vision



P1,1

P9,1

P9,9

P1,9

Figure 1: Basic interpretation of a pixel grid where the origin lies at the top left vertex, and y increases in a downwards direction.

One of the most prominent uses of computer vision in general is **object identification within an image**. There are 2 main features the human brain looks for when identifying an object in real life; **colour**, and **shape**.

Shape is arguably the most important aspect when thinking about classifying an object. This can be extracted by using **convolutional kernels to assess gradient changes in pixel intensity in comparison to their neighbour pixels**.

Images can be considered as matrices, with each individual pixel having a cartesian coordinate (x,y), and a pixel intensity value (z). The point of origin (0,0) on a computer monitor is always in the top left-hand corner of the screen, as shown in **Figure 1**.

# Loading images in R

When using **EBImage** you can read images with the function **readImage()**, which takes as input a file name or an URL. **EBImage** currently supports three image file formats: **jpeg**, **png** and **tiff**. For more specialised image types there is another complimentary library **RBioFormats** which provides support for a much wider range of file formats including microscopy image data and metadata. For more information on how to use **EBImage** you can click on the following link [EBImage-introduction](C://Users/e102010/Zotero/storage/MTTBCUFJ/EBImage-introduction.html).

To begin this practical, load a sample picture distributed with the package **EBImage**.

# f = system.file("images", "sample.png", package="EBImage")

# img = readImage(f)

When executing the commands in the box R loads the sample image into the R environment and names the new object **img**.

**Question: What class is the object img?**

# Displaying Images in R

To visualise the image (**img**) use the function **display()** and pass the name of the image as the first argument. Also, pass **method = "raster"** as the second argument. This will display the image using R’s build-in plotting facilities. The benefit of doing this is that it allows users to easily combine image data with other plotting functionality, for instance, add text labels etc.

The other way to display an image is by using the argument: **method = "browser"**.When called from an interactive R session, the **display()** function opens the image in a JavaScript viewer in your web browser. Using the mouse or keyboard shortcuts, you can zoom in and out of the image, pan, and cycle through multiple image frames.

To see how to browse through images with multiple frames load the image “**nuclei.tif**” from the **EBImage** package, which depicts cell nuclei.

nuc = readImage(system.file("images", "nuclei.tif", package="EBImage"))

Then display the image, using the argument **method = "browser"**,then use the arrows to move to the next image, as well as the zoom in / out buttons.

Until now we have only seen grayscale images, but R can also display and handle coloured images. To see an example, load the same sample image as above, but with colour and use the function **display()** to visualise it.

imgcol = readImage(system.file("images", "sample-color.png", package="EBImage"))

The coloured version of the sample image has **3 colour channels, or matrices**. To see the difference between the grayscale and coloured image let’s look more closely into the structure of each image. We can simply use the function **print()** passing the name of the image as argument.

print(img, short=TRUE)

Image

colorMode : Grayscale

storage.mode : double

dim : 768 512

frames.total : 1

frames.render: 1

Do the same for the **imgcol** and look at the output. It should look like the one below:

Image

colorMode : Color

storage.mode : double

dim : 768 512 3

frames.total : 3

frames.render: 1

As you can see, the two images, differ in the colorMode (Grayscale vs Color) and number of dimensions. To see the image dimensions you can also simply type **dim()** and pass the name of the image as argument. In the case of coloured images, you have a third dimension, corresponding to the 3 colour channels (RGB).

We can also, get more information regarding the structure of an **Image object**, by using the function **str()** . For example:

str(img)

Formal class 'Image' [package "EBImage"] with 2 slots

..@ .Data : num [1:768, 1:512] 0.447 0.451 0.463 0.455 0.463 ...

..@ colormode: int 0

..$ dim: int [1:2] 768 512

The **.Data** structure contains a numeric array of pixel intensities. Since this is a grayscale image the array is two-dimensional, with 768 times 512 elements, and corresponds to the pixel width and height of the image.

We can visualise the pixel intensities in the form of a table, using the **imageData()** function:

imageData(img)[1:3, 1:6]

Finally, the distribution of pixel intensities can be plotted in a histogram, and their range inspected using the range function.

hist(img)

Another way to obtain a summary of **Image** objects is by simply typing the object’s name.

Type **img** in the console and see what output you get.

**Task:** Repeat the same steps as above, for the **imgcol** image object, showing the pixel intesnsity arrays and histograms for each colour channel.

# Loading images from folders

Instead of just visualising images already available in a specific R package, we can also load and display images located in our computer or in the cloud, by providing the path to access the image. Use the **readImage()** function to load the image **strawb1.jpg** available in the Canvas page. If you have already placed the image in your working directory, you simply need to type:

strawb <- readImage("strawb1.jpg")

Then display the image.

**Task:** Use the steps shown above to check the structure of the image object **strawb** and visualise pixel intensities.

# Creating a Greyscale Image from RGB

Currently, the **strawb** image is a full colour image. **For the purposes of our practical we will only consider grayscale images.** The most common way to combine all 3 matrices to form a greyscale image is to take the average intensity of each pixel across all 3 channels. We could convert the image into grayscale manually, or simply use the function **channel()**, setting the argument mode to “grey” or “gray”. You can type **?EBImage::display** in the console to learn more about how this function works.

strawb.gray <-channel(strawb, mode="gray")

# Image Adjustments

At this point, there are two basic image manipulations that can be made. **Changing the image brightness or contrast can both decrease some minor noise changes, and accentuate certain edges.** Increasing the brightness is as simple as adding a value to each element of the matrix; be wary **when adding/subtracting values that all pixels of a greyscale image should stay in the range 0 to 1**. For example, try to increase the brightness of the stawb.gray image as below.

strawb\_bright = strawb.gray + 0.4

display( strawb\_bright )

To make sure we haven’t exceeded the maximum value of 1 for each pixel, we need to add a constraint.

strawb\_bright[strawb\_bright > 1] <- 1 # Replace values more than 1 with 1

**NOTE:** In case of subtracting a constant value from an image, we need to make sure we don’t end up with negative values. So in this case, we should add a constraint as follows:

strawb\_bright[strawb\_bright < 0] <- 0 # Replace values less than 0 with 0

Apart from changing the brightness we can take our **strawb.gray** image, and flip the bright areas to dark and vice versa by multiplying the image with -1.

strawb.inv <- normalize(-strawb.gray)

**NOTE:** The reason for using the function **normalise()** when inverting the image above, is to normalise it to the (0,1) range. Try without normalisation to see what you get.

We can also produce a negative image as above, by simply subtracting the image from its maximum value.

strawb\_neg = max(strawb.gray) – strawb.gray

display( strawb\_neg )

Changing the **brightness changes the overall intensity of the image, whereas contrast focuses on changing the intensity between the pixels themselves**. To change the contrast of an image we can multiply all the pixels it with a constant value. For example:

strawb\_contr = strawb.gray \* 2

display( strawb\_contr)

**Make sure the pixel intensity does not exceed 1!**

Finally, another image manipulation we can attempt, is **Gamma correction**. Gamma defines the relationship between a pixel's numerical value and its actual luminance. Without gamma, shades captured by digital cameras wouldn't appear as they did to our eyes. Our eyes do not perceive light the way cameras do. With a digital camera, when twice the number of photons hit the sensor, it receives twice the signal (a "linear" relationship). Instead, the human eyes perceive twice the light as being only a fraction brighter — and increasingly so for higher light intensities (a "nonlinear" relationship). See **Figure 2** for a graphical representation of the relationship between the actual luminescence and the detected light for humans and cameras. The reason for this difference, is that this non-linear relationship enables our vision to operate over a broader range of luminance. Otherwise, the typical range in brightness we encounter outdoors would be too overwhelming. The human eye is much more sensitive to changes in dark tones than changes in lighter tones since it allows vision to operate over a broader range of luminance.

**In a nutshell, gamma (or gamma correction) is what translates between our eye's light sensitivity and that of the camera**. When a digital image is saved, it's therefore "gamma encoded".

We can apply gamma correction to an image, through exponentiation. For example:

strawb\_exp = strawb.gray ^ 0.5

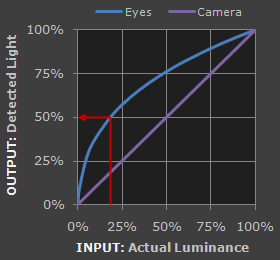
display(strawb\_exp)

Figure 2 Relationship between actual Luminance and light detected by human eyes and digital cameras.

Gamma is defined by Vout = Vingamma , where Vout is the output luminance value and Vin is the input/actual luminance value. This formula causes the blue line in Figure above to curve. When gamma<1, the line arches upward, whereas the opposite occurs with gamma>1.You can try different values for gamma correction and compare the results.

You can of course combine different operations in one line of code to create a fully adjusted image. However, it is good practice to look at, and compare, the adjustments made to the original. When applying image manipulations, we can use the function **combine()** to merge the individual images obtained after each manipulation into a single multi-frame image object.

strawb\_comb = combine(

strawb.gray, # original gray image

strawb.gray + 0.4, # brighter image

strawb.gray \* 2, # increased contrast

strawb.gray ^ 0.5 # gamma correction

)

Then to display all four images type

display(strawb\_comb, all=TRUE)

# Save images

Now that you have created different versions of the original image, you may want to save them for later use. It is common to use compression algorithms to reduce the storage consumption. There are two main types of image3 compression:

Lossless compression: it is possible to exactly reconstruct the original image data from the compressed file. An example for a storage format with lossless compression is PNG format.

Lossy compression: additional savings are made compared to lossless compression by dropping details that a human viewer would be unlikely to notice anyway. An example for lossy compression is the JPEG format.

For scientific images, it is good practice to store them in a lossless format. In the case of the strawberry plant image, we have been working with today, the original image was in a JPEG format. Let’s save the grayscale strawberry image in a JPEG format again.

writeImage(stawb.gray, "strawb1\_gray.jpeg", quality = 85)

The JPEG format allows to set a **quality** value between 1 and 100 for its compression algorithm. The default value of the quality argument is 100. When using a smaller value, this results in a smaller file size at the cost of some reduction in image quality. Similarly, we could have saved the image in PNG and TIFF format. Look into the help tab for more information on how to use the **writeImage()** function.

# Applying Linear (Gaussian) Filtering

Whilst changing the brightness and contrast can reduce the effect of some noise within the image, it is not an effective method. **One way to reduce noise is to spread the intensity over a series of pixels using a weighted matrix in a sliding window. For instance, we can define a window of a selected size around each pixel and average the values within that window**. After applying this procedure to all pixels, the new, smoothed image is obtained.

When applying a Gaussian filter for noise reduction (smoothing), the Gaussian function in 2D is defined as:

Where  **(sigma) denotes the spread of the intensity (standard deviation)**. This looks more complex than it is!

In **EBImage**, the 2-dimensional convolution is implemented by the function **filter2()**, and the function **makeBrush**() can be used to generate weight functions (filters).

w = makeBrush(size = 51, shape = "gaussian", sigma = 7)

strawb.smooth = filter2(getFrame(strawb.gray, 1), w)

In the code above, we used a Gaussian filter of width 7 given by sigma. The argument **size** is numeric and defines the size of the brush in pixels. This should be an odd number; even numbers are rounded to the next odd one. Type **?makeBrush** to see what other shapes are available.

**Task:** Use different size and sigma parameters to see the effect of different Gaussian filtering on the **strawb.gray image**. What do you observe?

If the filtered image contains multiple frames, the filter is applied to each frame separately. For convenience, images can be also smoothed using the wrapper function **gblur()** which performs Gaussian smoothing with the filter size automatically adjusted to sigma. For example, to apply Gaussian filtering in the **nuc** object you created earlier, containing multiple images of cells under microscope, type the following:

nuc\_gblur = gblur(nuc, sigma = 5)

display(nuc\_gblur, all=TRUE )

The use the arrows to browse through the individual frames.

In general, linear filtering can be used to perform both low-pass filtering (to blur images, remove noise, etc) and high-pass filtering (to detect edges, sharpen images).

# Edge Detection

Edges in images are areas with strong intensity contrasts; a jump in intensity from one pixel to the next. The process of edge detection significantly reduces the amount of data and filters out unneeded information, while preserving the important structural properties of an image. As mentioned above, in signal processing the operation of smoothing an image is referred to as low-pass filtering. High-pass filtering is the opposite operation which allows to detect edges and sharpen images. This can be done, for instance, using a Laplacian filter. Try different filters to see how the image changes.

lap.filt = matrix(1, nrow = 3, ncol = 3)

lap.filt[2, 2] = -8

strawb\_filt = filter2(strawb\_comb, lap.filt)

display(strawb\_filt, all=TRUE)

**Question:** Which transormation provides the best edge detection result?

# Image Thresholding

# Image thresholding is a simple type of image segmentation which partitions an image into a foreground and background. Image thresholding is most effective in images with high levels of contrast. We can simply apply global thresholding by setting a global threshold. This way we convert an image from colour or grayscale into a binary image, i.e., one that is simply black and white. For instance:

strawb\_thresh = strawb.gray > 0.5

display(strawb\_thresh)

Another type of thresholding is **adaptive thresholding,** in which the threshold is allowed to be different in different regions of the image. Adaptive thresholding works by comparing each pixel’s intensity to the background determined from a local neighbourhood. This can be achieved by comparing the image to its smoothed version, where the filtering window is bigger than the typical size of objects we want to capture.

This technique assumes that **the objects are relatively sparsely distributed in the image**, so that the signal distribution in the neighbourhood is dominated by background.

Let us first test adaptive thresholding on the nuclei image. For this image the assumption above is true since the nuclei are sparsely distributed.

disc = makeBrush(31, "disc")

disc = disc / sum(disc)

offset = 0.05

nuc\_bg = filter2(nuc, disc)

nuc\_th = nuc > nuc\_bg + offset

display(nuc\_th, all=TRUE)

**NOTE:** A faster way to apply adaptive thresholding is by using the function **thresh()**, which uses a faster implementation compared to directly using **filter2()**. Type the code below and check if you get similar results as before:

display(thresh(nuc, w=15, h=15, offset=0.05), all=TRUE)

**Task: a)** Apply global thresholding for the nuclei image and compare it to the adaptive thresholding above. Which type of thresholding works best at separating the objects from the background? **b)** Now try adaptive thresholding for the **strawb.gray** image. Compare it to the global thresholding results you obtained earlier. Which technique works best at separating the plant leaves from the background? Why?

# Image Segmentation

Image segmentation performs partitioning of an image and is typically used to identify objects in an image. Non-touching connected objects can be segmented using the function **bwlabel()**, while the functions **watershed()** and **propagate()** use more sophisticated algorithms able to separate objects which touch each other.

In the nuclei image example, although most nuclei are separated, some are close to each other and get merged into one big object even when applying adaptive thresholding as seen in **nuc\_th**. For this reason, we will try the **watershed()** function to overcome this issue. The watershed algorithm treats a grayscale image as a topographic relief, or heightmap. Objects that stand out of the background are identified and separated by flooding an inverted source image. In case of a binary image its distance map can serve as the input heightmap. The distance map, which contains for each pixel the distance to the nearest background pixel, can be obtained by the function **distmap**().

nmask = watershed(distmap(nuc\_th), 2)

display(colorLabels(nmask), all=TRUE)

In the course of this practical you have encountered some of the basic image analysis techniques used in computer vision. In many cases these techniques are used as image pretreatment methods with the aim of extracting features prior to applying Neural Network or Convolutional Neural Network modelling for image analysis. Other deep learning methods such as object detection techniques, including Region Proposal Convolutional Neural Networks (RCNN) and You Only Look Once (YOLO), are able to process images, without any preprocessing steps.

**Feel free to experiment with different parameter values in the above tutorial and see how the image quality is affected.**

**Optional Task:** You have also been provided with the image “apple.jpg” which shows a cut apple with an internal disorder. Use the tools above to try and isolate the affected areas in the apple from the “healthy” fruit. Are you able to successfully separate the areas?