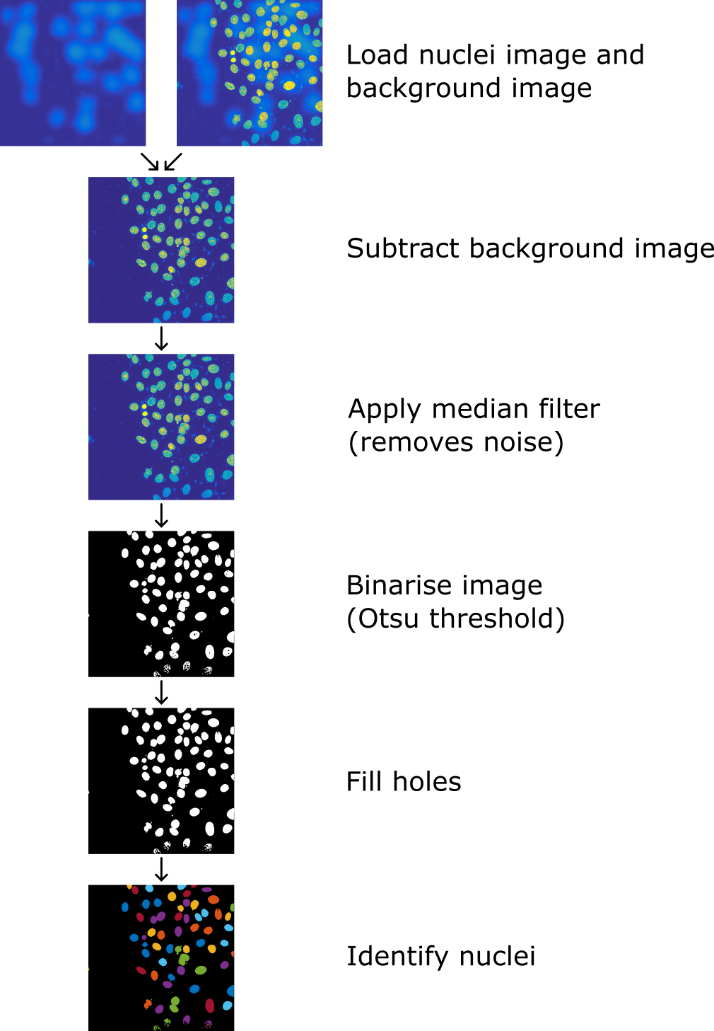
**MATLAB for image processing: Session 2 worksheet**

****In this worksheet we’re going to use what you’ve learnt about arrays, matrices and images to implement a basic image segmentation and analysis workflow. This workflow involves loading an image of cell nuclei, subtracting a background intensity profile (provided from a reference image), followed by binarizing the image and identifying connected regions (nuclei). The image to the right shows an outline of the steps you’ll be taking (Note: the first few lines use the parula colourmap to enhance the contrast). In an optional final exercise, you will measure the size of the nuclei in terms of the number of pixels they occupy in the image.

1. **Loading images from file**

The first step we need to take is to load the two images into the workspace. One shows the nuclei we want to segment, while the other shows the general background intensity profile. In a real situation you can get such background images when no sample is present, or for fluorescence microscopy, by imaging a sample with homogeneous fluorescence (e.g. a fluorophore solution).

1. Create a new script file, into which we will create our nuclei segmentation workflow. Save this file to a location accessible to MATLAB. Note: remember to start the script with the *clear* command.
2. Load the nuclei image to the workspace and assign it a reference. The image to use (“NucleiImage.tif”) is included in the Session 2 GitHub repository (<https://github.com/SJCross/MATLAB-course>).
3. Load the background image to the workspace and assign it a reference. The image to use (“BackgroundImage.tif”) is included in the Session 2 GitHub repository (<https://github.com/SJCross/MATLAB-course>).
4. Use *imshow* to display the two images.
5. **Subtracting the background**

We can use array operations to subtract the background image from the nuclei image.

1. Subtract the background image from the nuclei image and assign the output to a new reference.
2. Use *imshow* to display the background subtracted image and ensure it looks sensible. Background subtraction should remove the large bright patch, but leave the nuclei.
3. **Removing noise in the image**

Before segmenting the nuclei, we need to remove noise in the image. Failure to do this will result in lots of artefacts being identified as nuclei. A good filter for this purpose is a 2D median filter, which is “edge preserving” (i.e. it doesn’t tend to blur the image like a mean or Gaussian filter would). As with the worksheet from Session 1, you need to do some reading around (documentation, Google, etc.) to identify how to do this.

1. Identify and implement the relevant function to apply a 2D median filter (filter radius of 2px) to the image.
2. Assign the filtered image a new reference in the workspace (to differentiate it from the non-filtered image).
3. Use *imshow* to check the filtered image looks correct (the filtering step shouldn’t alter the image much, just smooth the texture a bit).
4. **Binarising the image (calculating the threshold)**

The next step is to binarise the image, so the nuclei are shown in white and the background in black. This can be represented as a logical image, where all elements are either true (1) or false (0). For this, we first need to calculate the intensity threshold above which pixels are classed as corresponding to “nuclei”. The *graythresh* function uses the method of Otsu [1] to determine the threshold based on the intensity histogram for the image.

1. Use the *graythresh* function to calculate an intensity threshold for the median-filtered image. Store this value in the workspace with an appropriate reference. Note: The threshold is output in the range 0-1. Check the data type for the median-filtered image and scale this threshold accordingly.
2. **Binarising the image (applying threshold using loops)**

Now we have a threshold value, we can binarise our image. We will do this two ways: one using loops, the other using array operations. Doing this will demonstrate the difference in performance of the two methods. Here, we will do the binarisation using loops. The aim is to iterate over each pixel in the median-filtered image and set the pixel value as logical true (1) or logical false (0) depending on whether the pixel intensity is greater than the threshold, respectively.

1. Initialise a logical array of equal size to the median-filtered image.
2. Use loops to iterate over each pixel in the median-filtered image, check if it’s greater than the threshold and assign the corresponding value in the logical (binary) image. Note: You can nest loops (i.e. have one inside the other) to iterate over two or more dimension (in this case, x and y).
3. Use the *tic* and *toc* commands to measure the time taken to iterate over all pixels.
4. Remove the logical array initialisation step and run the process again, recording the new execution time. You should see the execution time increase compared to when the logical array had been initialised.
5. Use *imshow* to look at the output binarised image. Based on a quick inspection, does the binarised image look sensible compared to the input image?
6. **Binarising the image (applying threshold using array operations)**

Next, we will disable the loop-based thresholding and replace it with an equivalent operation using array operations. As with exercise 5, we will measure the time taken for this operation to complete (we should see a decrease in the time taken). Note: In normal usage you would probably use the *im2bw* or *imbinarise* functions; however, we’re trying to learn array operations, so you’re not allowed to use these for this exercise!

1. Use the *%* character to comment-out the logical array initialisation and binarisation steps introduced in exercise 5.
2. Use array operation(s) to create the binarised image logical array. Hint: logical tests on arrays were covered briefly in the slides.
3. Use *tic* and *toc* to measure the execution time for this new method of binarising the image.
4. Use *imshow* to look at the new binarised image. It should look the same as the image you got using loops.
5. **Optimising the binarised image (filling holes)**

You’ll notice some of the nuclei in the binarised image have a lot of holes. Before we identify the nuclei, we want to get rid of these.

1. Search for a function to remove holes in binarised images. Apply this function and create a reference to the output image.
2. Use *imshow* to check the hole filling has worked correctly.
3. **Identifying nuclei objects**

We’re now ready to detect the nuclei in our binarised image. This can be done using a process called “connected components labelling”, whereby all groups of pixels in direct contact with each other are assigned a common numeric “label” (effectively an ID number for that nucleus). The pixels considered to be in “direct contact” are defined by a value called “connectivity”, which is either 4 (neighbours are immediately to the left, right, above and below) or 8 (same as before but including diagonals).

1. Use the *bwlabel* function to generate a labelled image, where each pixel’s intensity value is equal to its “label” (or nucleus ID number). You can leave connectivity as the default value (8). Note: background pixels (i.e. those not in a nucleus) will be assigned a label of “0”.
2. Store the labelled image with an appropriate reference.
3. Use *imshow* to display the labelled image. Note: before displaying the image, you may wish to play around with the *label2rgb* function, which will convert a labelled image into a colour image.
4. **[Optional extension exercise] Measure the number of pixels in each object**

This last exercise will likely be a bit trickier, so don’t worry if you can’t complete it, we’ll go through the solution at the start of Session 3. The aim here is to count the number of pixels in each object. For the output, it’s sufficient to just print a line such as “Object 12 has 546 pixels” to the command window for each object. We will cover data structures for storing results in more detail in Session 3. There are many ways to do this, so I won’t include a list of steps.

**References**

[1] Otsu, N (1979), "A threshold selection method from gray-level histograms", *IEEE Trans. Sys., Man., Cyber.* **9** 62-66