Exercise 1

Classification and detection

Make sure to save any functions that you create for the lab sessions. They might be useful. Exercises marked with a $\star$ are typically somewhat harder. They are never necessary to be able to follow the rest of the course, so my advice is to keep them for last.

Getting organized

Ex 1.1 Create a folder where you can place everything concerning these exercises and the lab sessions. We will assume you put it in your home folder and name it `ssy097`. To keep organized, it might be a good idea to create a new folder for each set of exercises.

Ex 1.2 Download the `stuff_for_ex_1` folder from the course page. Put it in `ssy097`.

Ex 1.3 Start Matlab and change working directory to the new folder either using the `cd` command or using the graphical interface. It might be a good idea to add `ssy097` and all its subdirectories to the Matlab path. You can do this either using the graphical interface or commands `addpath(genpath(x))`, where you replace `x` for the full path to the `ssy097` folder.

Getting help

Writing `help imread` in Matlab gives you a not-so-short description of how the function `imread` works. Using `doc imread` opens a new window with a longer help text. If you don't know the name of the Matlab function yet, try Google or the lab assistant.

Scripts

Whenever you do anything that requires more than two lines of code in Matlab it is a good habit to create a script for it, even if you are not planning to repeat it more than once. The built-in Matlab editor is a good option. Just type `edit` in Matlab to start it, or use the New button or Open button in the interface.

In this file you just list the Matlab commands that you would have run in the command window, one at each row.

Basic image handling

Ex 1.4 Open the editor and start writing a script. First, load an example image using

```
raw_image = imread('lemon.png');
```

The semi-colon is used to suppress output. Run the script using the buttons in the Matlab editor or by typing its name in the prompt.

Ex 1.5 Images are typically stored as `uint8` with integer values in $[0, 255]$ or `uint16` with values in $[0, 65535]$. It's more convenient to work with floating point values in $[0, 1]$. Convert the image to double using

```
img = im2double(raw_image);
```

Ex 1.6 Draw your image in a plot window using `imagesc(img)`. Also try viewing the individual color channels. You can select the kth channel using `img(:,:,k)` How blue is a lemon?

Ex 1.7 Try to plot an asterisk `*` in the bottom left corner of the image. You will have to use `hold on` to avoid clearing the figure and `plot(icol,irow,'*')` to plot the asterisk. Note the order that you give the coordinates in.

Ex 1.8 Convert the image to grayscale. There are different ways to do this but a simple one is to compute the mean of the 3 channels. This can be done with the `mean` command. `mean(img, 3)` will compute the mean over the third dimension, i.e., the color channels.

Ex 1.9 To correctly plot a grayscale image you use `imagesc(img)` followed by `colormap gray`. If you like you can write them on one row

```
imagesc(img), colormap gray
```

Functions

The next thing to try is to create a function. Unlike a script, a function takes a number of arguments but apart from these it cannot use the variables that you have defined in your workspace. To create a function `read_image` you create a file `read_image.m`. To show that this is a function and not a script the first line should look something like this:

```
function img = read_image(path_to_file)
```

This specifies that the function takes an input argument `path_to_file` and produces an output `img`. Note that the variable names are long but informative. Names such as `a` or `x` are fine in mathematics, but normally not suitable for programming.

Ex 1.10 Make a function

```
function img = read_image(path_to_file)
```

that takes the path to an image file and performs the steps in Ex. 1.4 and 1.5. Use this function whenever you want to load a color image.

Ex 1.11 Make a function

```
function img = read_as_grayscale(path_to_file)
```

that calls `read_image` and then converts to grayscale as in Ex 1.8. Use this function whenever you want to load a grayscale image. (You will save a lot of time if you actually follow this advice!)

Simple classifiers

Ex 1.12 Run `load bloodcells/cell_data.mat`. This loads a struct `cell_data` with a set of patches containing centered blood cells. Use

```
imagesc(cell_data.fg_patches{7})
```

to view the 7th patch. There is also a set of other patches from the same microscopic images found in

```
cell_data.bg_patches
```

Construct a linear classifier that separates the two sets as well as you can. Go through all the given patches and compute how many errors your classifier does.

One problem in the next exercise is to load K images without writing K lines of code. Here's one hint

```
for image_nbr = 1:100
image_name = ['img_' num2str(image_nbr) '.png'];
```

Another way is to use the `dir` command in Matlab.

```
dir('trafficsigns/speed/*.png')
```

will give you a list of all the png-files in the folder `trafficsigns/speed`.

Ex⋆ 1.13 In the `trafficsigns/speed` folder you find a set of 51×51-images of speed signs and in the `trafficsigns/bg` folder you find a set of 51×51-images containing essentially anything but speed signs. Try to construct a linear classifier that separates the two sets as well as you can. It is easier if you use all three color channels. Go through all the given images and compute how many errors your classifier does.

Ex⋆ 1.14 A linear classifier consists of a weight image w with the same size as the images that you want to classify and a threshold τ. Given a set of training images and a fixed w, can you suggest an efficient method to find the best τ.

Sliding window

Ex 1.15 Load one of the images in `bloodcells/test_images` using `read_as_grayscale`. It contains a microscopic image with a lot of blood cells. In order to count the blood cells, apply your linear classifier in a sliding window manner. To make this efficient, use `imfilter` for the sliding window and then threshold the whole response image at once. Visualize the result as suggested below. You might want to change the threshold to decrease the number of false detections. What is problematic about the way that you generated the original threshold?

One way to visualize the response is to draw the image in subplot and the classification result in another.

```
subplot(121)
imagesc(img)
colormap gray
subplot(122)
imagesc(result)
colormap gray
```

Another option is to mark the classified pixels with yellow in the original image (as you saw in the lecture). You can do this with the provided `view_with_overlay` function. Assuming that `result` has ones at pixels classified as cells you run

```
view_with_overlay(img, result);
```

Non-maximum suppression

The response that you just viewed cannot be used directly to count the blood cells as there are multiple high-response pixels at the same cell. Next, we will fix this using non-maximum suppression.

Ex 1.16 Explore performing max filtering with `ordfilt2` over 3×3-neighbourhoods. Also try median filtering. What is the effect of median filtering.

There are many ways to solve the next exercise. If you are new to Matlab, here are some facts that might help you out.

```
indicator = (image1 == image2)
```

will create an image with ones at pixels where `image1` and `image2` have the same values and zeros otherwise.

```
[row_coords, col_coords] = find(img1 > img2)
```

will output the coordinates of all pixels where `img1` is strictly larger than `img2`.

Ex 1.17 Write a function

```
maxima = strict_local_maxima(image)
```

that computes the coordinates of all *strict* local maxima in `image`. Hint: Use `ordfilt2` with 3×3-neighbourhoods. Let the output `maxima` be a $2 \times n$-array. Since this is Matlab standard, let the first row be the column coordinates and the second row the row coordinates.

Ex 1.18 Generate a simple test case for `strict_local_maxima.m` and verify manually that it is treated correctly. You could, for example, generate a small random image using

```
img = randi(10,5,5)
```

This will create a 5×5-image with integer elements between 1 and 10. Verify manually that the correct maxima are detected.

Ex 1.19 Try `strict_local_maxima.m` on the image produced by `img = zeros(30,30)`. If you get any maxima, then your function is not working properly.

The detector... finally

Ex 1.20 Make a function `centres = cell_detector(img)` that uses you linear classifier together and non-maximum suppression to detect cell centres in an image `img`. Try it on a few of the images in `cells/test_images`, that is, run the detector to produce a set of positions and then plot these positions in the image to see if they are right. How well would it work to count the cells?

Ex⋆ 1.21 Make a similar detector for speed signs and try it on one of the images in

```
trafficsigns/test_images
```

If you used all color channels you cannot do sliding window directly with `imfilter`. Do it one color channel at the time instead. It's a good idea to create a function

```
score = sliding_window(img, w);
```

for that. How well does the detector work? Don't expect too much.. We've just had one lecture.

Exercise 2

Mixed exercises

Ex 2.1 Figure 2.1 shows two images of John Lennon. The same images as

`john_small.png` and `john_big.png`

on PingPong. On the left, we have a low-resolution image. It could, for example, have been obtained at a large distance. (It also appears to be slightly out of focus.) On the right we have a high-resolution variant that has been filtered with a Gaussian. The two images can be seen as sampled from the same scale space image $L(x, y, \sigma^2)$, but the distance between sample points is 5 times larger in the low-resolution image.

In the lectures, we computed gradient histograms for these images and concluded that they looked similar. But if we want to compare the absolute length of gradients we have to be careful.

a. If we use the a filter

$$f = (-0.5 \quad 0 \quad 0.5) \tag{2.1}$$

to compute horizontal derivatives in the two images. How do we need to rescale them to get similar values? If you like you can load the images and verify your result.

b. When we computed gradient histograms in Lab 1, we computed one gradient per pixel. How will the number of pixels at the two resolutions affect the absolute values in the histogram?

c. We use the gradient histograms for our SIFT-like descriptor and the last step in computing the SIFT-like descriptor is normalization. So do we really need to bother about **a** and **b**?

Figure 2.1: Paul at a distance left (left) and a close-up after Gaussian filtering (right).

The next exercise is concerned with subpixel precision. You can use any of the detectors that you have designed or the example given in the file `subpixel.mat`. It contains a full-resolution image `img`, a low-resolution variant `low` and a simple linear classifier (`w`, `thr`) for detecting motion capture markers in the low-resolution image. The idea is to detect the markers in the low resolution image with sub-pixel precision. It is 625 times faster than using the full-resolution image.

Ex⋆ 2.2 Make a function

```
delta = subpixel_max(patch)
```

that takes a 3×3-patch as input. Assuming that the centre point of this patch is a local maximum, the function determines the offset of the true local maximum with subpixel precision.

Use your function to refine the local maxima produced by the provided `w` and `thr` when applied to the image `low`. Multiply the resulting coordinates with 5and plot them in the full-resolution image. Also try with the unrefined local maxima (without subpixel precision).

In the next exercise, we want to compute a sliding window standard deviation. To do this efficiently in Matlab we want to use built-in functions such as `imfilter`. First note that

$$I \cdot I = I_{sq} \cdot \mathbb{1} \tag{2.2}$$

where I_{sq} is the result of squaring all the elements in I. Now

$$Var(I) = \frac{1}{N_{el}} (I - \mu_I \mathbb{1}) \cdot (I - \mu_I \mathbb{1}) \overset{*}{=} \frac{1}{N_{el}} I \cdot (I - \mu_I \mathbb{1}) = \frac{1}{N_{el}} I_{sq} \cdot \mathbb{1} - \frac{1}{N_{el}^2} (I \cdot \mathbb{1})^2 . \tag{2.3}$$

The $*$-equality is proven in (2.9) in the lecture notes and the final equality follows from the definition of μ_I, see (2.7) in the lecture notes. This result is the analogue of

$$Var(X) = E[X^2] - (E[X])^2, \tag{2.4}$$

in statistics. This allows efficiently computing the variance in Matlab using a combination of `imfilter` and element-wise operations such as `.*` or `.^`

Ex⋆ 2.3 Make a function

```
std_img = local_std(img, patch_size)
```

that computes a sliding window standard deviation over all patches of size `patch_size`. Try to make your implementation efficient by using `sliding_window` from Ex. 1.21.

Ex⋆ 2.4 Try your `local_std` on images created with `randn(100, 100, 3)`. What standard deviations do we expect to see? What about `5*randn(100, 100, 3)`?

Finding patches with maximum correlation is very useful to detect small movements or perspective changes in video.

Ex⋆ 2.5 Take your classifier from Ex. 1.21 or load the provided in `trafficsign_classifier.mat`. Your `w` should already have mean 0. Rescale it so it also has standard deviation 1. Using

```
sliding_window(img, w) ./ local_std(img, patch_radius) / numel(w)
```

you can compute a sliding-window correlation with w. Try to find a new threshold to turn this into a classifier. Test it on the images in `trafficsigns/test_images` Does it work better than the linear classifier?

Exercise 3

Mixed exercises

Ransac and camera geometry

Ex 3.1 Consider estimating an affine transformation for image registration. Assuming that the rate of outliers is 90%, work out how many Ransac iterations you need to do perform to get approximately 100 outlier-free estimates. More precisely we want the expected number of outlier-free estimates to be 100.

Ex 3.2 A common digital SLR camera has 4000×6016 pixels, a sensor size of 15.4×23.2 mm and a focal length of 105 mm (at full zoom). Consider two points with pixel coordinates $(3008, 2000.5)$ and $(3009, 2000.5)$. How many degrees are there between the corresponding rays in 3D? You can assume that the principal point is at the image centre, i.e., at $(3008.5, 2000.5)$.

Ex 3.3 Given a camera matrix

$$P = \begin{pmatrix} 100 & 0 & 50 & -350 \\ 0 & 100 & 50 & -250 \\ 0 & 0 & 1 & -3 \end{pmatrix}, \tag{3.1}$$

compute the camera position, C, (in world coordinates). Hint: Look at Section 10.2 in the lecture notes.

Ex 3.4 Compute the projections of

$$U_1 = \begin{pmatrix} 8 \\ 12 \\ 5 \end{pmatrix} \quad \text{and} \quad U_2 = \begin{pmatrix} 8 \\ 12 \\ 3 \end{pmatrix} \tag{3.2}$$

in the camera of equation (3.1). What is the depth of U_2? What happens when you try to compute the pixel coordinates of U_2?

Ex⋆ 3.5 Consider a problem where we require five measurements to estimate the parameters of a model. Assume that there are 75% outliers in the data. What is the probability that after 100 Ransac iterations, we have never sampled an outlier-free set. After how many iterations is this probability less than 1%.

Iteratively reweighted least squares⋆

For affine and similarity transformations we can solve for a least squares solution by using \ on a system $M\theta = b$. In some cases we want to weight the different residuals, for example, if we know that some measurements are more uncertain. This is easy, we just multiply

$$\bar{r} = M\theta - b \tag{3.3}$$

with a diagonal matrix, W, with the square root of the weights on the diagonal. This creates a new system $M'\theta = b'$ with $M' = WM$ and $b' = Wb$, that can be solved just as easily.

Recall from Section 8.6 in the lecture notes how we can use weighted least squares to minimize the Huber loss. Start by solving a least squares problem. Then create a weight matrix such that the elements of the ith residual are weighted with

$$w_i = \begin{cases} 1 & \text{if } |r_i| < \delta \\ \frac{\delta}{|r_i|}, & \text{otherwise.} \end{cases} \tag{3.4}$$

and solve the weighted least squares problem. Note that W should contain the square roots of the w_i's! Can you see why? Use the solution to compute new residuals and new weights. Iterate. Note that in (3.4), the r_i refers to the current estimate of r_i, so it is to be considered as a constant when solving the least squares problem. (Otherwise it wouldn't be least squares.)

A disadvantage with using Ransac followed by least squares is that it is a little sensitive to the choice of outlier threshold. In `irls_affine.mat` you find an example of this. `pts` and `pts_tilde` are the inliers after Ransac with a large outlier threshold. In `A_ls` and `t_ls` you find the resulting transformation. If you compare the resulting `warped_ls` to `tgt` you see the poor alignment. In the next exercise we will try to improve this using the Huber loss.

Ex⋆ 3.6 Implement iteratively reweighted least squares for affine transformation estimation. You can set the number of iterations to 5 and $\delta = 3$. Test it on the data in `irls_affine.mat` Warp the provided `src` image using the resulting transformation and see how similar to `tgt` you can get it.

Filtering

Ex⋆ 3.7 In `guess_the_filter_1.mat` there is a small image `img` and the response of filtering this image with an unknown 3×3 filter (with integer elements). Try to estimate the filter. In `guess_the_filter_2.mat` the problem is somewhat harder since moderate noise has been added to the response map. Does your method still work?