Image Analysis, Laboratory session 4

Starting up

Download the file `session4.zip` from the course home page. Extract it, e.g. by right clicking and choosing Extract here. This will create a directory called `session4`, which contains all the scripts and images you need for this session. Start MATLAB and see to it that `session4` is the current working directory. (`pwd` and `cd` might help.) Next load the data sets for this session by running the script

```
>> session
```

1 Clustering

1.1 K-means

Open `kmeans.m` in a text editor and study the code. Fill in the missing code to calculate new cluster centres and run with three clusters on the points specified in `Px` and `Py` or `Points`.

2 Fitting

2.1 Hough transform

A method capable of detecting lines even if they are partially hidden, is the Hough transform.

Exercise: You have already loaded the images `lines`, `dots` and `illusion`. Study the image `lines`, for example using

```
>> imagesc(lines); colormap(gray)
```

Look at `dots` in the same way. Note that both images are binary, meaning that only one bit per pixel is required to store the images. Now explore the MATLAB command `hough`. Write e.g. `help hough`. Test it on the two images images using

```
>> [transformed_image,theta,rho]=hough(image);
```

and interpret the result. To get the axes right show the transformed image using

```
>> imagesc(theta,rho,transformed_image)
```

Note that the horizontal axis corresponds to the line orientation and the vertical to its perpendicular distance to the origin. Compare with the images in Forsyth & Ponce. You might want to threshold the images. Which points in the Hough transform corresponds to which lines in the original image?

Now look at the image `illusion`. There is no complete triangle in the image, but our brain interprets the fragments as one. Now we will check if the computer is also capable of finding the broken lines.

First find the lines in the image, e.g. using the Canny edge detector. Then apply the Hough transform on the result. Can you find the three triangle edges in the transform? Which other lines can you find?

Feel free to try the Hough transform on an edge image from the lab session on edge detection.

2.2 Line fitting

Write a MATLAB function that performs line fitting using one of the methods from the course, (e.g. least squares or total least squares). Try your function on the data points in `Qx` and `Qy` or `Qpoints`.

2.3 Fitting of multiple lines using k-means

Forsyth & Ponce describes an algorithm (15.2) to fit a number of lines to points. Modify `kmeanslines.m` so as to use your line fitting from above when fitting a line to points. The output from your line fitting function should be a matrix l of size 3×1 with the line parameters

$$l = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \quad \text{as in} \quad ax + by + c = 0.$$

Consequently the first line of your function definition should be something like

```
function l = myfitline(data);
```

Try to fit three lines to the points with coordinates `Rx` and `Ry` (also in `Rpoints`).

2.4 Line fitting with RANSAC

Forsyth & Ponce includes an algorithm (15.4) for line fitting when a large part of the data points are outliers. Study the script, `ransaclines.m` and try using it on the points with coordinates `Rx` and `Ry` . As earlier it is expected that there exists a function `myfitline` that fits one line to a set of points.

3 Computer Vision

Computer vision is a very popular field of research today. Some central problems are mapping, localization and classification. The goal is to get computers to perform some of the simple tasks we do every day, like recognizing people or objects or finding our way in a city or a wood or a store room. You will now study some very simple versions of these problems.

First of all, we assume that we are able to identify a set of especially interesting points. It could be corners, or light sources or something else which is easy to detect automatically.

Exercise: Run the MATLAB script `bakmovie`. It illustrates a central relation in computer vision. Which?

We also assume that the world we are studying is two-dimensional and that the camera takes one-dimensional images. Thus the interesting points are points in the plane and the images can be seen as a pencil of beams around the camera centre.

3.1 Localization

Exercise: Run the MATLAB script `trylaser`. It shows a realistic imaging situation. By clicking in the plot you will get an illustration of a measurement (an image) from this location. Quit the script by clicking in the figure window margin, i.e. outside the axes.

Since the measurements differ depending on the location of the camera, it is often possible to determine the camera position from an image.

Exercise: Running `simplelocate` will show the same map and a measurement (an image). Try to move the *camera* and rotate the beam pencil to fit the points. Use the left and right mouse buttons to rotate the beam pencil, and the middle button to move the camera. To help you, one of the whole line should go through the point marked with a circle. Quit by clicking in the margin.

It is rather difficult to find the right parameters by hand, but theoretically, three beams and their corresponding points are sufficient to determine the camera position. Due to measurement errors it is essential to use as many beams as possible.

When the number of visible points are bigger than three, the locali<ation problem has to be formulated as an optimization problem. The optimization problem can then be handled with methods from our optimization course. What would be a suitable goal function? Why?

3.2 Recognition and localization

It is rather easy to perform localization when you have a map and knows which image point correspond to which point on the map. It is much more difficult if you only have the map.

Exercise: Running `difficultlocate` shows you the map again and yet another image. Try to move and rotate the beam pencil as in the previous example. This time no correspondence is known.

Matching image points to a map using just the one-dimensional measurements is rather difficult. One way to handle the problem is to simply keep testing different correspondences, but that can take a lot of time and if there is a lot of noise it is hard to know if you have found the best solution. Another approach is to use some invariant characteristic in the image. If four of the

visible points lie on a line, one such invariant is the cross ratio

$$\frac{\tan(\alpha_1) - \tan(\alpha_3)}{\tan(\alpha_2) - \tan(\alpha_3)} \Big/ \frac{\tan(\alpha_1) - \tan(\alpha_4)}{\tan(\alpha_2) - \tan(\alpha_4)}$$

where α_1, α_2, α_3 och α_4 are the measured angles to the colinear points. This ratio does not depend on the position of the camera.

Exercise: Running `trycrossratio` shows you the map again. At the four points their measured angles are displayed and at the bottom of the plot the calculated cross ratio. Try moving the camera around and note that the cross ratio does not change. This invariant can be used to identify specific four point configurations. Quit the script by clicking in the margin.

If you have a map of, e.g. a room, the cross ratio for all relevant sets of collinear points can be calculated once and for all and stored in a table. For a given image the cross ratio can then be calculated for interesting groups of four points and compared to the table. This is repeated until enough information is gained to locate the camera.

Exercise: Run the script `autolocate` for a demonstration on how the cross ratio can be used for recognition. Note the explanations in the plot window and ask the teaching assistant if anything is unclear.