DM432 Computer Vision Term Paper

Alex Dănilă

March 2009

Contents

Chapter 1

Binary Regions

1.1 Image Segmentation

1.1.1 Region labelling on a binary image

Region labelling is the process of identifying the sets of pixels which are interconnected, each set representing a *region*. Interconnected pixels, in the case of binary images, are those pixels that are neighbouring and have the same colour. All the pixels in a region are assigned a number that is unique to that region (a "label") , and this number will facilitate further segmentation operations on the image. There exist different methods for performing region labelling, but in the present section only *flood filling* will be presented. Also, in order to perform region labelling the definition of neighbouring must be chosen, a pixel has either 4 or 8 neighbours in the simplest cases.

Region labelling with flood filling. The idea of flood filling algorithms is the following: search for a pixel that is not labelled, create a new label for it, and give the same label to each of its neighbours. Flood filling can be done in many ways, three of them are: recursive, iterative breadth-first, iterative depth-first.

The recursive version of the algorithm is the simplest to think of, and can be implemented in a minimum of lines of code, but has proved to work only on the smallest images, in all other cases the execution stopping with a stack overflow because of the big number of recursive calls.

The iterative versions of the flood filling algorithm require the explicit definition of a list of points in the picture that need to be processed. At the beginning of the algorithm the first pixel is inserted in the list, and then the list is processed in different ways depending on the algorithm version. In the breadth-first version, the first pixel in the list removed from the start of the list, labelled, and all its neighbours are added at the end of the list for processing (in this case the list acts like a queue, *First In, First Out* manipulation). In the depth-first version, the last pixel is poped from the list, labelled, and all its neighbours are added at the end of the list (the list acts as a stack, *Last In,*

Figure 1.1: Region labelling on a simple image containing 5 objects.

First Out manipulation)[22].

The 8 neighbours version of the flood filling has proved to be very useful in the case of complex images, where the regions are not obvious at all to the naked eye, as can be seen in the image 1.2.

1.2 Convex Hull and Diameter of a Region

1.2.1 The convex hull of a region

The *convex hull* or *convex envelope* is the minimal convex polygon that contains all the points from a set. As intuitive description, the convex hull of an object can be imagined as wrapping an elastic band around an object in order to encase it.

The computation of the convex hull in a two dimensional space can be done with many different algorithms, like: Jarvis march, Graham scan, the divide and conquer algorithm for the convex hull, Chan's algorithm[10]. For spaces with more than two dimensions the task is much harder; an example is Chan's algorithm, which also works for three dimensional spaces[10].

Graham scan algorithm is a popular algorithm for computing the convex hull that runs with a time complexity of $O(n \log n)$, n = number points considered. Description:

1. Find **P** - the point with the lowest coordinate on the y axis. In the case that there are more points with the same y coordinate, choose the one

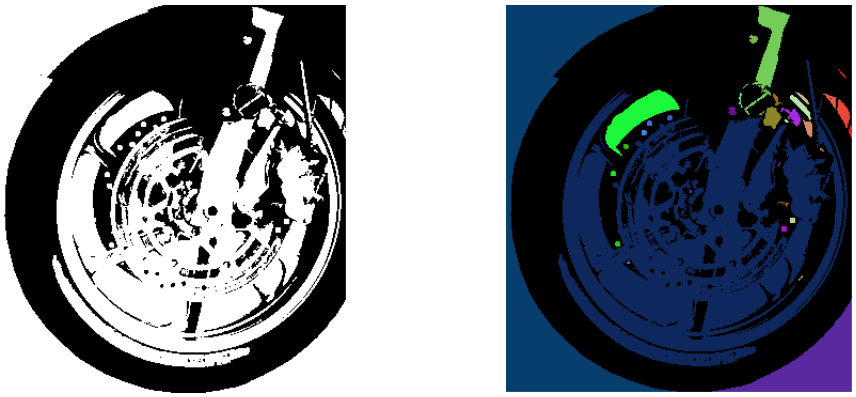

Figure 1.2: Region labelling (8 neighbours) shows regions which are not obvious to the naked eye otherwise.

with the smallest coordinate on the x axis.

2. Compute the angles that the other points and P make with the x axis using the following method:
-consider the segment from P to another point to be the vector $d_x d_y$;
-depending on the value of $d_x$choose the value of the angle to be:

$$\begin{cases} arctangent(d_y/d_x) & if\ d_x > 0, \\ \pi & if\ d_x = 0, \\ \frac{3}{2} - arctangent(d_y/(-d_x)) & if\ d_x < 0. \end{cases}$$

3. Sort the set points of region in increasing order of the computed angles.

4. Create a list of points that will form the hull. Try to add to the hull list each of the points from the sorted list of points. When the last three points in the hull list form a right-turn (a concavity in the hull), remove the last point until there is no other concavity. Continue until reaching the starting point.[11] See image 1.3 for a graphical illustration of how the algorithm works from Wikipedia.[1]

The points considered for computing the convex hull should be only the ones from the contour of the region; using all the points of the region becomes too computationally expensive even for medium sized images because the sorting of the angles is applied to all the points of the image. For example, using a hand written insertion sort algorithm, the convex hull of the picture 1.4 took 3 seconds to compute, on the full-size original image (1021 x 806 pixels), the computation was still running after 2 hours. With the Collections.sort method

[1] http://en.wikipedia.org/wiki/Graham_scan - Wikipedia page on Graham scan

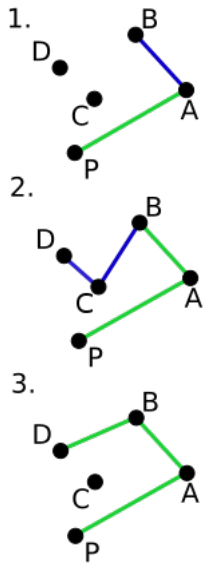

Figure 1.3: Graham scan illustration, as found on Wikipedia.Starting from point P, points A, B and C are added to the hull in order, and the rotation between them is clockwise. When trying to add D to the existing $PABC$ list, the rotation from BC to CD is counterclockwise. C is discarded and the search continues with D. The algorithm continues like this until point P is reached and there are no counterclockwise turns.

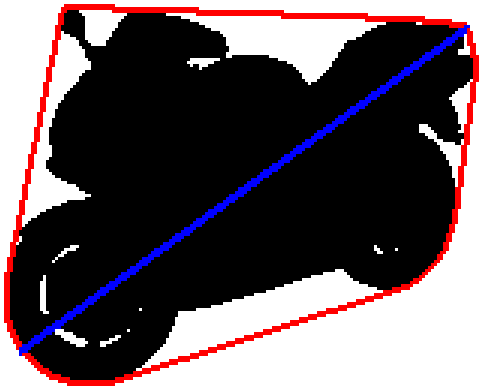

Figure 1.4: Convex hull and maximum diameter of a motorcycle shape.

provided by Java, the first image would be processed instantly, and the original one in around 5 seconds. Working only with the contour points makes the operation fast even for big images.

1.2.2 Area of the convex hull

The area of the convex hull can be approximated using the Gaussian area formula for non-self-intersecting polygons (simple polygons). The formula is as follows:

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i).$$

In the formula$x$ and y are coordinates of the points that describe the contour of the polygon; the first and last points are the same (this means $x_n, y_n = x_0, y_0$). In order to apply the formula, the points of the polygon must be ordered clockwise of counterclockwise; if they are ordered clockwise the computed area will have negative value but will be correct in absolute value. In our case of computing the hull area, the points are ordered clockwise.

1.2.3 Maximum diameter of a region

The maximum diameter of a region is a set of two points from the region for which distance is maximal. The current algorithm for identifying the points is computing the distance between each pair of points from the hull, thus having a complexity $O(n^2)$ (the actual number of computations is $n * (n-1)/2$). This complexity is a concern only for very large images.

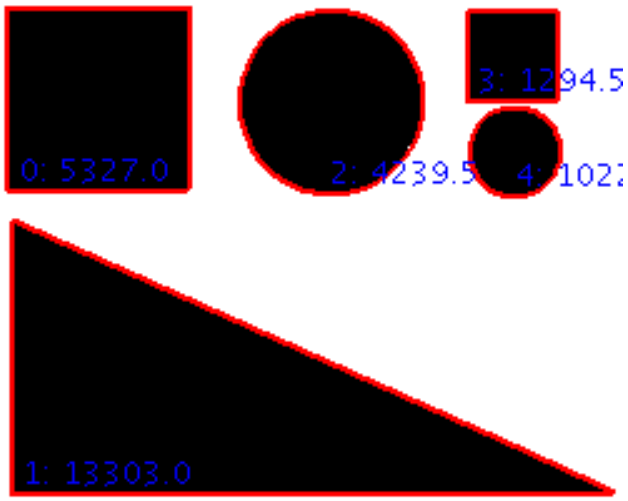

Figure 1.5: Simple objects and their convex hull area (expressed in pixels) computed using the Gaussian area formula. The ratio between the area of a square and the enclosed circle can be verified for the two the circles and squares pairs, for example objects 3 and 4: $A_{circle} = \frac{\pi}{4}d^2 \approx 0.7854 \cdot d^2$, $\frac{A_4}{A_3} = \frac{1022}{1294.5} = 0.7894$.

Chapter 2

Binary Regions - Region Moments

2.1 Central moments

Central moments are numerical measures describing the placement of pixels around its centroid. The **centroid** of an image region (similarly applies to all geometrical figures), also called geometric centre or barycentre, is the intersection of all straight lines that would divide the region into two regions with equal areas. The centroid can be computed as the average of its points coordinates, and is similar to the *centre of mass* of an object: if you cut a cardboard shape of a region, the centre of mass of that shape would be the same as its centroid in the image processing sense.[8, 22]

Moments in image processing can be described as a statistical property of the distribution of the pixels in a region[3]. *Central moments* are denoted with μ_{pq}, where p and q are numbers to be used as exponents of horizontal and respectively vertical distances. Some central moments are: *second moment*-μ_{11} called variance (a measure of statistical dispersion), *third moment*-μ_{22} called skewness (a measures of the symmetry of the statistical distribution), fourth moment-μ_{33} called kurtosis (a measures of how sharp the peak in a statistical distribution is).[7] Pseudocode for computing the central moment μ_{pq} of a region is presented in algorithm 2.1.

Algorithm 2.1 Pseudocode for computing the central moment of a region.

for each pixel in region:

μ_{pq}+=(pixel.column-centroid.column)^p·(pixel.row-centroid.row)^q

Given the way that the central moment is computed, it's value is dependent on the scale of the image. To make the central moment scale invariant

(**normalised central moment**, denoted by $\overline{\mu}_{pq}$ or by η_{pq}) we divide the value of the central moment by

$$s^{(p+q+2)/2},$$

where $s \in \mathbb{R}$ is a chosen factor[15]. The correctness of this value can be verified experimentally, as can be seen in image 2.1, but to demonstrate it mathematically is harder. Intuitively, the computation used for finding this value, should be based on the following:

1. in algorithm 2.1, the scale parameter s is multiplied as follows:$s^p \cdot s^q$for each pixel.
2. since the image is a two dimensional object, the number of pixels added has a quadratic growth (the number of pixels is proportional to the square of on dimension of the image), thus making the s parameter multiplications to grow quadratically.

Normalised central moments are translation and scale invariant, but not rotation invariant.

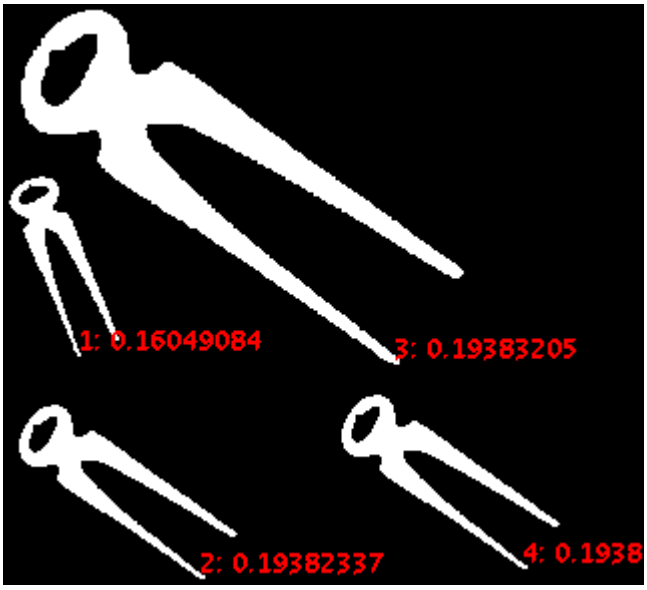

Figure 2.1: Four instances of the same objects and their normalised centre moments ($\overline{\mu}_{11}$-variance). The normalised centre moments are invariant under translation and scaling, but not under rotation.

2.2 Hu's moments

Because normalised central moments are not rotation invariant a reformulation is necessary. One method to produce rotation invariant moments is using **Hu's**

moments, which are values computed from normalised central moments (up to order three), as follows[12, 15]:

$$
\begin{aligned}
I_1 =& \overline{\mu}_{20} + \overline{\mu}_{02}, \\
I_2 =& (\overline{\mu}_{20} - \overline{\mu}_{02})^2 + (2\overline{\mu}_{11})^2, \\
I_3 =& (\overline{\mu}_{30} - 3\overline{\mu}_{12})^2 + (3\overline{\mu}_{21} - \overline{\mu}_{03})^2, \\
I_4 =& (\overline{\mu}_{30} + \overline{\mu}_{12})^2 + (\overline{\mu}_{21} + \overline{\mu}_{03})^2, \\
I_5 =& (\overline{\mu}_{30} - 3\overline{\mu}_{12}) \cdot (\overline{\mu}_{30} + \overline{\mu}_{12}) \cdot [(\overline{\mu}_{30} + \overline{\mu}_{12})^2 - 3(\overline{\mu}_{21} + \overline{\mu}_{03})^2] + \\
& (3\overline{\mu}_{21} - \overline{\mu}_{03}) \cdot (\overline{\mu}_{21} + \overline{\mu}_{03}) \cdot [3(\overline{\mu}_{30} + \overline{\mu}_{12})^2 - (\overline{\mu}_{21} + \overline{\mu}_{03})^2], \\
I_6 =& (\overline{\mu}_{20} - \overline{\mu}_{02}) \cdot [(\overline{\mu}_{30} + \overline{\mu}_{12})^2 - (\overline{\mu}_{21} + \overline{\mu}_{03})^2] + \\
& 4\overline{\mu}_{11} \cdot (\overline{\mu}_{30} + \overline{\mu}_{12}) \cdot (\overline{\mu}_{21} + \overline{\mu}_{03}), \\
I_7 =& (3\overline{\mu}_{21} - 3\overline{\mu}_{03}) \cdot (\overline{\mu}_{30} + \overline{\mu}_{12}) \cdot [(\overline{\mu}_{30} + \overline{\mu}_{12})^2 - 3(\overline{\mu}_{21} + \overline{\mu}_{03})^2] + \\
& (3\overline{\mu}_{21} - \overline{\mu}_{03}) \cdot (\overline{\mu}_{21} + \overline{\mu}_{03}) \cdot [3(\overline{\mu}_{30} + \overline{\mu}_{12})^2 - (\overline{\mu}_{21} + \overline{\mu}_{03})^2].
\end{aligned}
\tag{2.1}
$$

In figure 2.2 a series of objects with different transformations (rotation, scaling, mirroring) and in table 2.1 the values for the Hu's moments can be compared ($\log_{10}$ has been applied to the values to make the range of values smaller). The values of the moments are indeed invariant, but they are sensitive to approximation errors, the moments 6 and 7 are particularly sensitive, as much as to be no longer rotation invariant for small regions, a fact tested on diverse images.

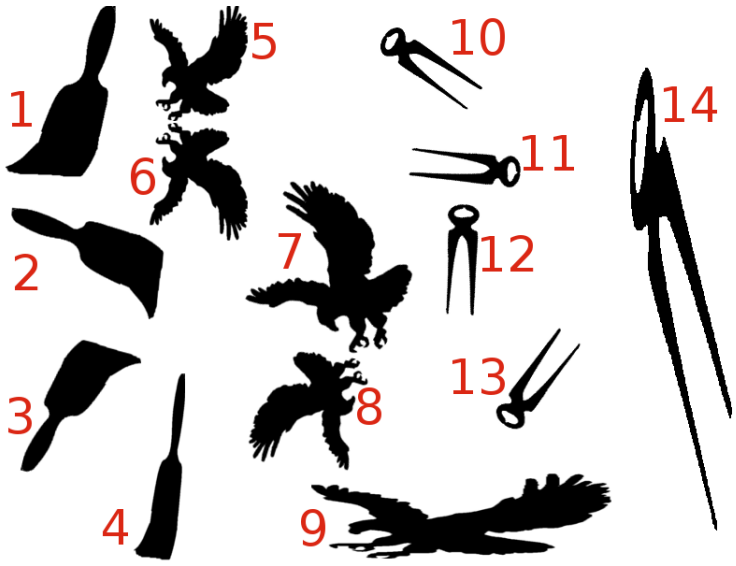

Figure 2.2: Objects, rotated, mirrored, scaled uniformly and non-uniformly, to test the values of Hu's moments.

	1	2	3	4	5	6	7
1	-0.45	-1.08	-1.67	-2.02	-3.88	-2.79	-4.2
2	-0.45	-1.08	-1.67	-2.02	-3.87	-2.81	-3.89
3	-0.45	-1.08	-1.66	-2.01	-3.86	-3.48	-3.76
4	-0.16	-0.36	-0.82	-0.89	-1.75	-1.13	-2.97
5	-0.59	-2.2	-2.47	-3.32	-6.22	-4.91	-6.22
6	-0.58	-2.18	-2.48	-3.31	-6.2	-4.88	-6.24
7	-0.58	-2.2	-2.47	-3.29	-6.18	-4.77	-6.12
8	-0.59	-2.21	-2.47	-3.3	-6.19	-5.07	-6.43
9	-0.28	-0.66	-1.76	-2.44	-4.84	-4.42	-4.79
10	-0.26	-0.74	-1.61	-1.4	-2.9	-2.28	-2.88
11	-0.26	-0.74	-1.59	-1.38	-2.87	-1.75	-5.17
12	-0.26	-0.75	-1.61	-1.4	-2.9	-1.77	-2.96
13	-0.26	-0.74	-1.6	-1.39	-2.88	-3.73	-2.62
14	0.05	0.06	-0.44	-0.4	-0.82	-0.5	-0.46

Table 2.1: *Hu's moments* for the objects in the image 2.2. On the rows the object number, on the columns the moments from 1 to 7. The values are $\log_{10}$.

2.3 Which of the Hu's moments to use

From the 7 different moments, one might want to choose only one for discriminating between different region shapes. One idea for choosing one of moments is to test with many visually different images and see what is the correspondence between variation of moment value and the visual difference between the different region shapes. Making this kind of visual inspection the 6^{th}moment revealed bigger differences between different objects and smaller between more similar objects. This can be seen in the example image 2.3 and the corresponding table 2.2. For example objects 1 and 5 have the most similar shape, and that is the place where the 6^{th}moments have the smallest differences as number. The 1^{st}moment has a small range of values, while the 7^{th}moment continues to have big deviations, although in this case the shapes had a medium size. The other moments kept showing small differences between visually different shapes in both this and other tests.

Another measure of the differences could be done using the *standard deviation* of the moments' values. In a series of tests, considering shapes that have big differences, the Hu moment with the biggest deviation could be chosen. The same technique could be applied on shapes that are similar, this time choosing the smaller deviation. Of course, like the first method presented, this would also suffer from ambiguity in defining what is the difference between two shapes. This measure can be improved if instead of simple standard deviation we use a *relative standard deviation* (*coefficient of variation*[1])[9].

[1]More at http://en.wikipedia.org/wiki/Coefficient_of_variation

Figure 2.3: Different objects used to test correspondence between shapes and Hu's moments values.

	1	2	3	4	5	6	7
1	-0.26	-0.6	-1.31	-1.43	-2.8	-3.19	-3
2	-0.64	-2.37	-2.84	-2.87	-5.73	-4.06	-5.73
3	-0.58	-2.2	-2.47	-3.3	-6.19	-4.9	-6.19
4	-0.69	-2.07	-3.06	-4.75	-8.66	-6.19	-10.36
5	-0.45	-1.08	-1.67	-2.02	-3.87	-2.78	-4.46
6	-0.62	-2.94	-3.13	-3.25	-6.44	-8.12	-6.75
St. dev.	0.14	0.79	0.69	1.05	1.88	1.83	2.27

Table 2.2: Hu's moments values ($\log_{10}$)for image2.3, together with the standard deviation of the values (St.dev.) on the last row.

Chapter 3

Geometry and Straight Lines

3.1 Introductory exercise

Given two points in the 2D plane

$$x_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \; x_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$

compute the normal form of the line that passes through the two points . This translates into a system of two equations:

$$\begin{cases} ax_1 + by_1 + c = 0 \\ ax_2 + by_2 + c = 0 \end{cases} .$$

This system has one tree variable but only two equations, so we consider $a = 1$in order to solve it. The solution to this system of equations is:

$$\begin{cases} a = 1 \\ b = \frac{x_2 - x_1}{y_1 - y_2} \\ c = -x_1 - y_1(\frac{x2 - x1}{y_1 - y_2}) \end{cases} .$$

Thus, the equation of the line that passes through points (0, 10), (200, 100) is

$$x + \frac{-20}{9}y + \frac{200}{9} = 0.$$

After normalisation the values become: $a = 0.41$, $b = 0.912$, $c = 9.12$. The plotted line can be seen in figure 3.1.

3.2 Plotting an Algebraic Line in ImageJ

On a graphics device, an algebraic line needs to be represented as a segment that touches the edges of the visible area. In order to draw a segment in ImageJ

(as in most other graphics tools as well), the line needs to be defined by two points contained by the line. For drawing this segment, the set of points of the line that touch the edges will be computed. Considering the equation of the wanted line to be $ax + by + c = 0$, the intersections with the following lines will be computed:

$$0x + 1y + 0 = 0 - the\ top\ of\ the\ device\ (Ox\ axis)$$
$$0x + 1y + height - 1 = 0 - the\ bottom\ of\ the\ device$$
$$1x + 0y + 0 = 0 - the\ left\ side\ of\ the\ device\ (Oy\ axis)$$
$$1x + 0y - width + 1 = 0 - the\ right\ side\ of\ the\ device.$$

This computation does not raise special problems and is done straightforward with the following formula for computing the intersection point P(x, y) of two lines $a1 \cdot x + b1 \cdot y + c1 = 0$ and $a2 \cdot x + b2 \cdot y + c2 = 0$:

$$x = (b1 \cdot c2 - b2 \cdot c1)/(a1 \cdot b2 - a2 \cdot b1)$$
$$y = (c1 \cdot a2 - c2 \cdot a1)/(a1 \cdot b2 - a1 \cdot b1)$$

In the case that $(a1 \cdot b2 - a2 \cdot b1) = 0$, the computation will not be performed and this is not a problem, as it means that the two lines are parallel. There is no problem in loosing 2 points when the line is parallel with Ox or Oy, as there will exist the other two points of the line that intersect the other coordinates axis.

After computing these points, we will see if they are on the screen edges (that is, a point is contained by one of the segments that represent the screen edges). There can be: 0, 2, 3, or 4 points after this validation step. If there are more than 2 points it means there exist some duplicates that can be removed. Two small notes need to be made here:

- a)there is no problem in drawing all possible segments between all the points in the case that they are more that 2, as the segments will be identical;
- b)if there are 2 points and they make a duplicate, the duplicate should not be removed, as in this case the line is intersecting only one corner of the device.

Attention need to be paid to the fact that in Java there exist two values for 0, +0 and −0. Having two values for 0 meant that the comparisons I did in order to see if a point is contained by a screen edge could possibly give a false negative (the point was computed not be on the segment while it was). Take the following example:

> Consider that the computed point was at ($x = -0$, $y = 0$). When trying this point on the Ox axis: "$if(x >= 0) and (x < width)$" the result would be that x is not contained by the segment, while it actually was. Of course, this could have been avoided in the first place by taking all the values back to the integers field.

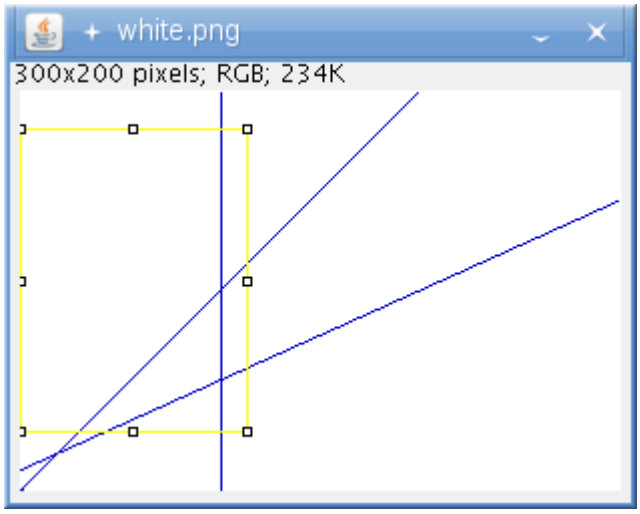

Figure 3.1: ImageJ, plotting the following lines: $x + \frac{-20}{9}y + \frac{200}{9} = 0$ (the line from the first exercise), $x + y + 0 = 0$ (the line of slope 1 that passes through the origin) and $x + 0y + 100 = 0$ (a vertical line that passes through the point (100, 0).
Note:
In ImageJ image processors, like in other painting devices, the origin of the coordinate system is in the top left corner and the Oy axis is oriented down. In this image the Oy axis is reversed and the origin of the coordinate system is translated to the bottom left corner in order to make the visualisation intuitive, something not done is other circumstances because is complicates debugging.

Chapter 4

Line fitting

4.1 Total-least-squares fitting

In the context of this assignment, total-least-squares fitting is a method of finding the straight line that best approximates a set of 2D points. It's idea is to minimise the sum of squared distances from the line (L) to the points (set P of N points, $P = \{p_1, ..., p_{N-1}\}$, which means minimising[16]:

$$\sum_{i-0}^{N-1} d^2(p_i, L) = \sum_{i=0}^{N-1} (ax_i + by_i + c)^2.$$

4.1.1 Implementation

The implementation has the following steps:

1. From the image, create a list of points that are of interest for fitting the line.
2. Compute the values of $\overline{x}, \overline{xx}, \overline{xy}, \overline{y}, \overline{yy}$.
3. Using the provided EigenSolver to find the parameters a, b, c of the best fitting line.
4. Plotting the line.[16]

Below are the results of running the implemented plugin. To date, because on defects that I did not find, it is not functioning correctly. Given this state of the implementation, except for test images, no further details have been examined. As can be seen in the images 4.1,4.2 and 4.3, the plotted line does not fit the points. In these tests the line is more vertical than it should be, but there are also case where it is horizontal but should be vertical. A comparison between the wrong the lines found by the two algorithms can be seen in figure 4.4.

Intuitively, this method should have as a big disadvantage that outliers weight very much (given that the method optimises the sum of squared distances).

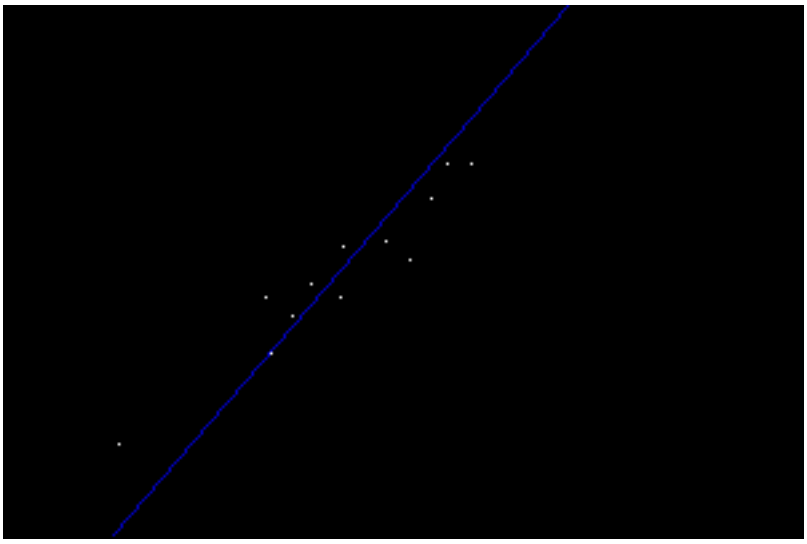

Figure 4.1: Total-least-squares test 1, wrong currently, $a = -1092.4, b = -936, c = 230876$.

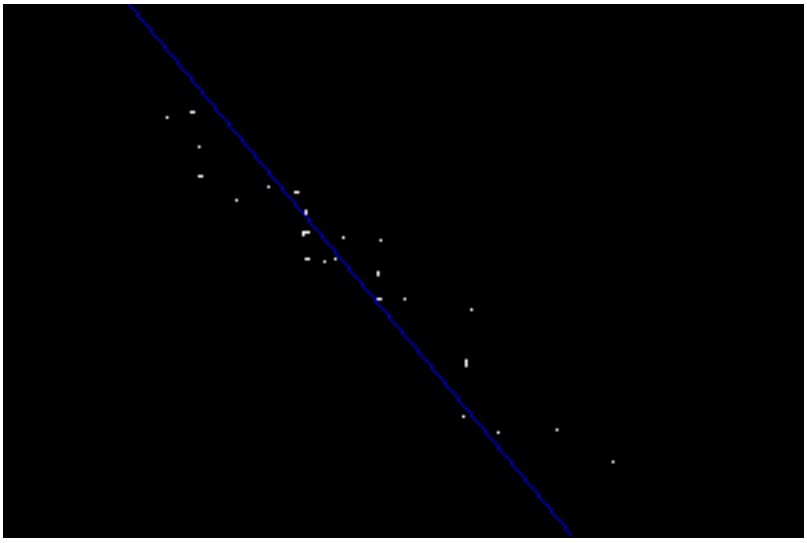

Figure 4.2: Total-least-squares test 2, wrong currently,$a = -1551, b = 1290, c = 74115$.

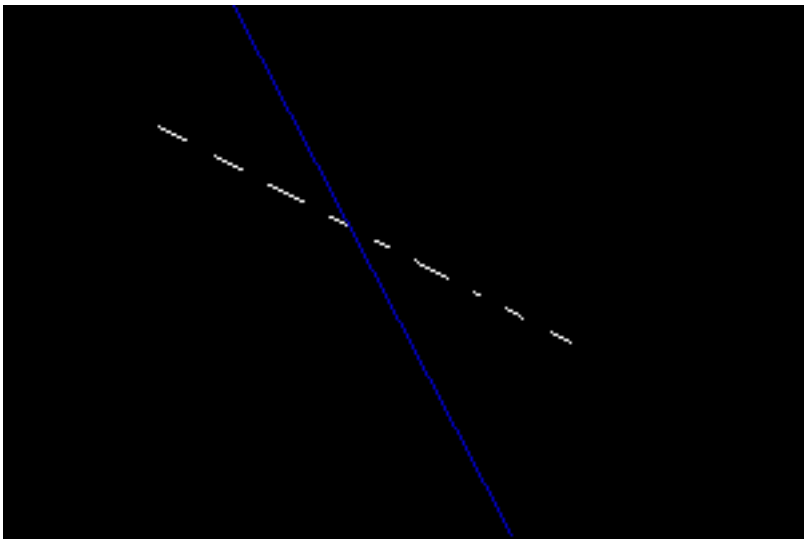

Figure 4.3: Total-least-squares test 3, most obviously wrong from all, $a = -2178, b = 1143, c = 189072$.

4.2 Least-median-squares fitting

Least-median-squares is another algorithm for finding the line that best fits a set of points. It's idea is to find a line such that median of the squared distances from the line L to the points is minimum.

4.2.1 Implementation

The implementation repeats the following steps, until a satisfying line is found[18]:

1. Choose two points p_1, p_2 from the set of points.
2. Compute the line that passes through the p_1 and p_2.
3. Compute a list of squared distances (S) from the line to the points.
4. Sort S and find the value of the median squared distance S_M. Remember the values p_1, p_2, S_M if S_M is the smallest until now.
5. If S_M is smaller than a threshold or the maximum number of steps has been reached stop, otherwise go back to step 1.

The results of this algorithm are very good, but, as in other cases where a mean is required, it requires additional data structures for the list of values, and many computations for sorting.

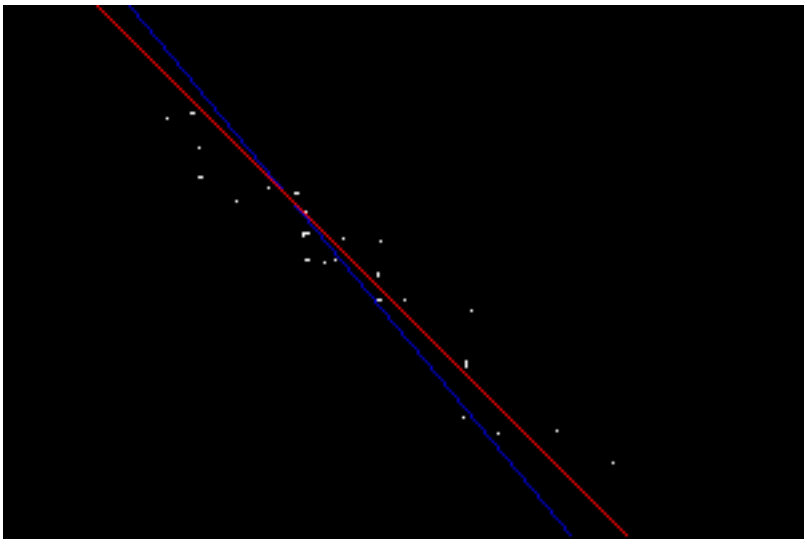

Figure 4.4: Test image 2, showing the difference between the incorrect line found by the total-least-squares algorithm (blue) and the one found by the least-median-squares (red)..

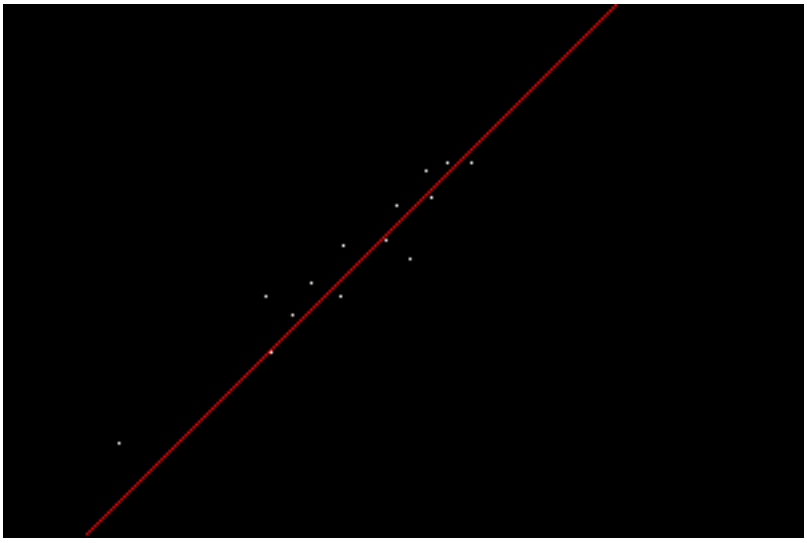

Figure 4.5: Least-median-squares test 1, $a = 1, b = 1, c = -229$, the coordinate system is the default one of the device (the origin is in the upper left corner, and the Oy axis direction is downwards).

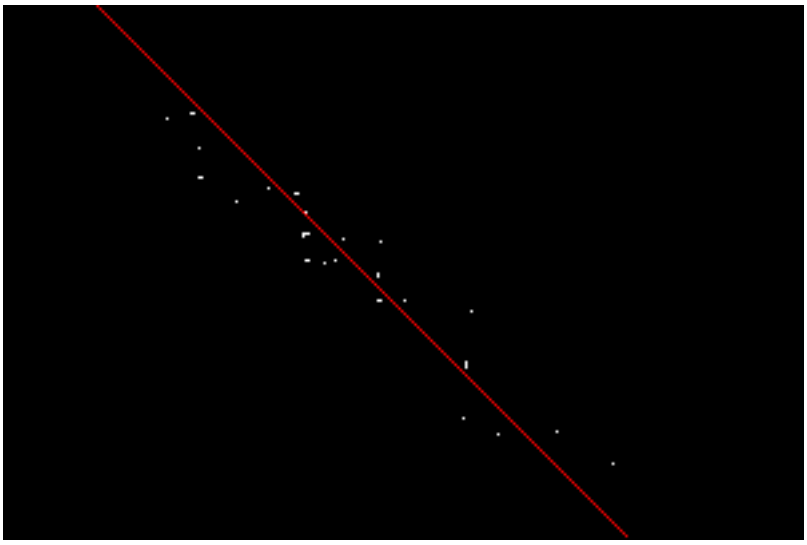

Figure 4.6: Least-median-squares test 2, $a = 1, b = -1, c = -35$.

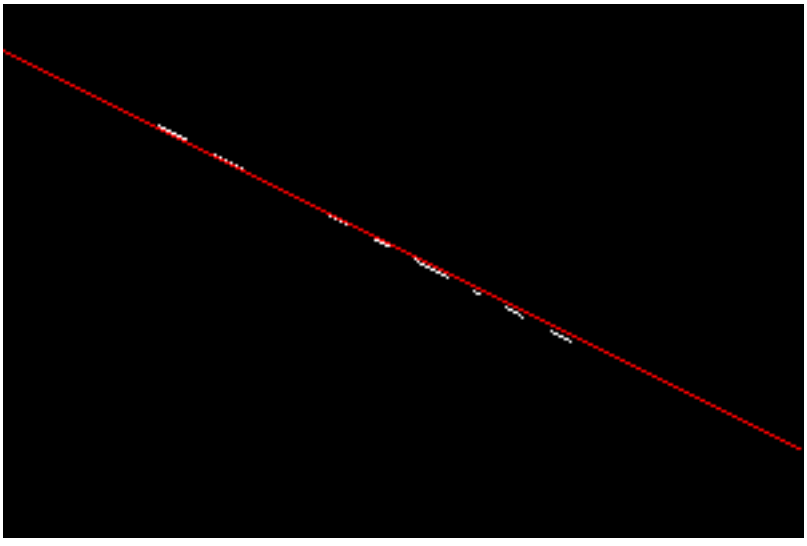

Figure 4.7: Least-median-squares test 3, $a = 1, b = -2, c = 34$.

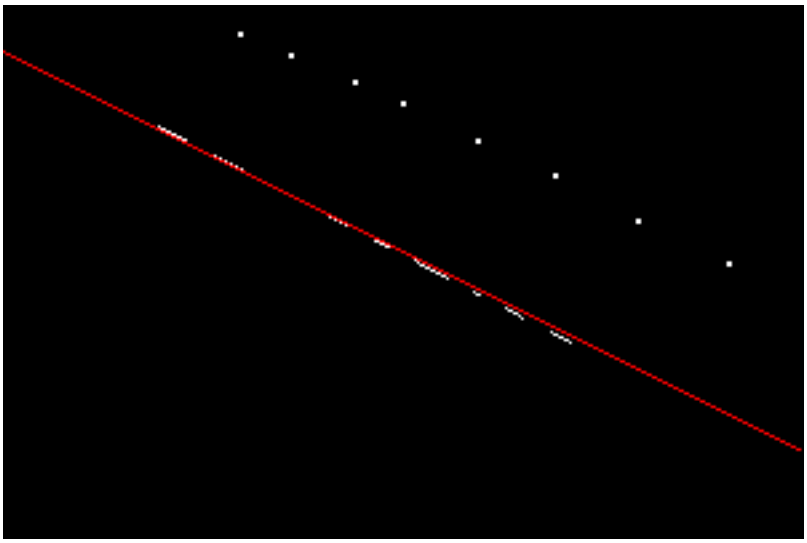

Figure 4.8: Inserting a small number of outliers in test image 3 does not change the line at all.

4.2.2 Adding outliers

The least-median-squares algorithm proved to deal successfully with outliers in all reasonable cases. It always follows the trace with the biggest number of points and it is actually hard to make the line leave the trace. One example of doing this can be seen image 4.9. It is actually hard to make the line leave a well defined line, even in the presence of many outliers.

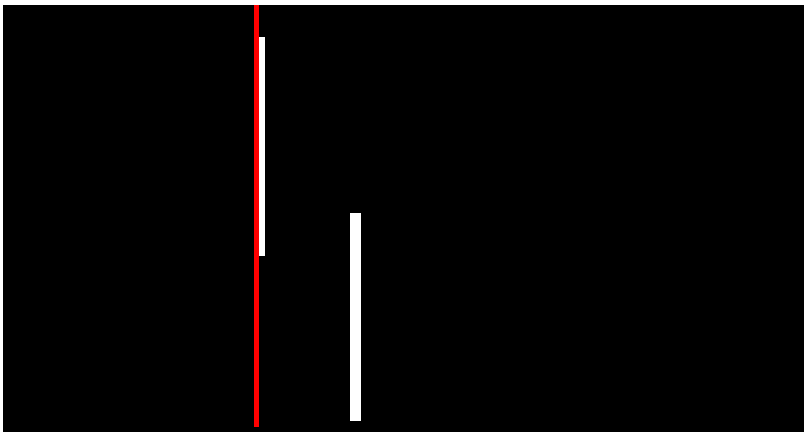

Figure 4.9: The line sticks to the block with more pixels (2 pixels more the block on the left block). It doesn't even go 1 pixel to the right, where it would be more equilibrated.

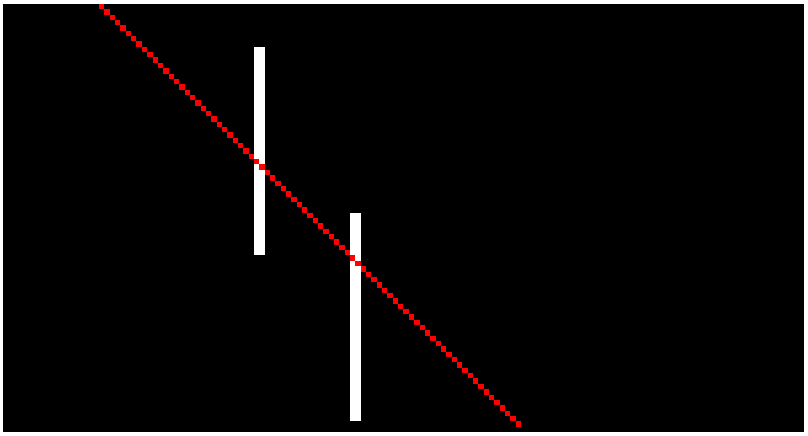

Figure 4.10: Two blocks with the same number of pixels. This time the median cannot choose one group with more pixels that the other.

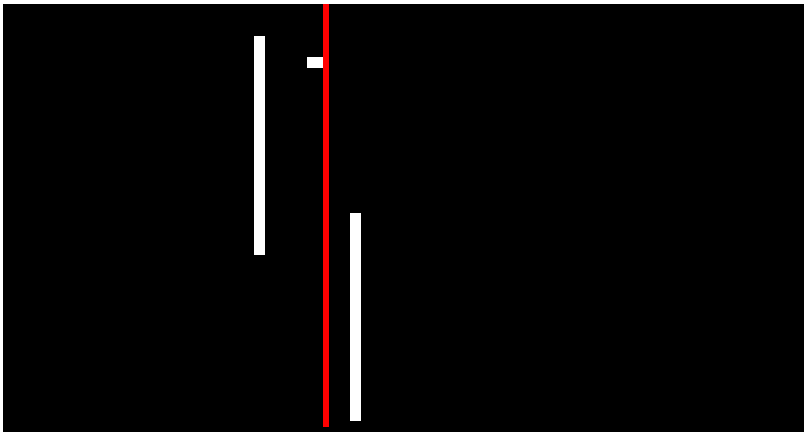

Figure 4.11: One last example where the line leaves the biggest trace of pixels.

Chapter 5

Marker Recognition and Localisation

Marker recognition is the operation of discovering predefined physical objects in a picture in order to draw conclusions either about the position and orientation of that object in space, or to draw conclusion about the position of the camera.

ARToolKit is a computer vision tracking library that allows of applications that overlay virtual imagery on the real world. To do this, it uses video tracking capabilities in order to calculate the real camera position and orientation relative to square physical markers in real time. Once the real camera position is known a virtual camera can be positioned at the same point and 3D computer graphics models drawn exactly overlaid on the real marker.[5]

An ARToolKit physical marker is a white coloured board with a black drawing enclosed in a black square. The drawing inside the square has a shape and positioning that makes it recognisable from different angles. The black drawings on white board make the markers have a good contrast.

The purpose of the current assignment is to recognise the four corners of typical ARToolKit markers. The main parts of the algorithm are selecting regions from the image that are candidates for being markers, tracing the contours of the regions and fitting four lines of the regions contour.

5.1 Algorithm

The algorithm work has the following main steps:

1. Thresholding the image to isolate regions that are candidates for being ARToolKit markers.

2. Selecting regions that cannot be ARToolKit markers and tracing their contours.

3. Fitting four lines on the region's contour and intersecting them (the most important step).

The detailed steps are as follows:

5.1.1 Isolating the candidate regions

The image is inverted such that the threshold is applied a usual for selecting the brightest object. The a simple threshold is applied. The middle of the RGB interval (255/2) was appropriate for all test images. The Thresholding result can be

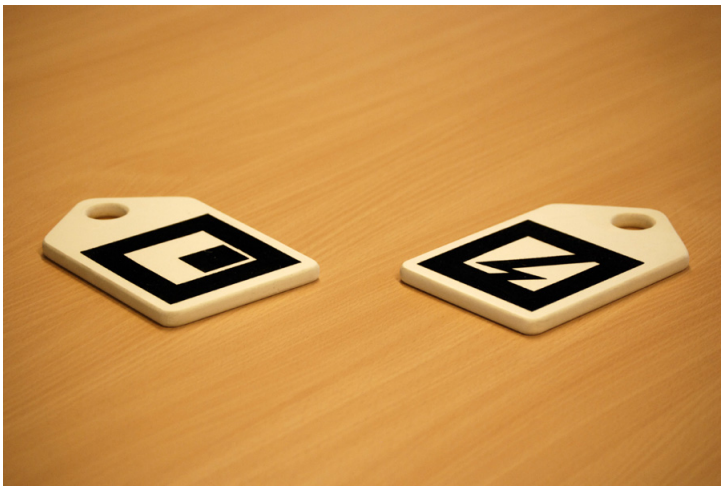

Figure 5.1: Original marker image.

5.1.2 Selection the candidate regions and drawing the contour

The candidate regions are selected using the following constraints:

- a regions should contain at least 1% and at most 75% of the pixels in the image. The limits are somewhat arbitrary, but they eliminate tiny regions (and possibly large poorly lighted zones, but this never happened in test images.
- a region should not extend on more than 99% of the width or height of the image. This eliminated the darkened exterior of the images and would insure that the marker is completely visible.
- a region should not be contained in another region, this eliminated the internal drawing of the markers, but would pose a problem in real applications if a large object would be in the background of the marker.

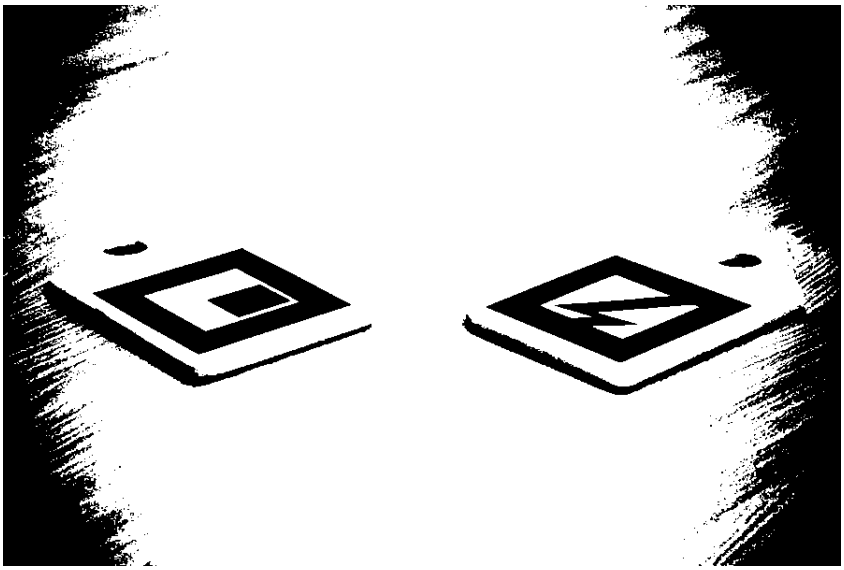

Figure 5.2: Image 5.1 after the simple thresholding.

Figure 5.3: After the selection steps are applied to image 5.2 no other regions than the markers remain.

After having the candidate regions (image 5.3), the outer contours are traced using the contour tracer from source code supplied with the book Digital Image Processing [22]. These points will later be used for the line fitting.

5.1.3 Fitting the lines on the contour and intersecting them

An algorithm for fitting more lines on a set of pixels was developed, having the following coarse steps:

1. Start with a random line and compute the distances from the line to the points;
2. Create new line by fitting 25% of the closest points using least square fitting. The distances are computed a again this step repeats until a number of points are very close to the line.

Figure 5.4: The contours (scaled), after processing image 5.3.

3. Optionally, for the noisy images, take the points that are closest to the line from the previous step and do a least medians medians fitting.

4. Remove the points that are closest to the line just found and go to step 1.

5.1.3.1 Detailed description of the incremental line fitting algorithm:

1. A line with random coordinates is created and the distances from the line to each of the points is computed.

2. Iterative step. The distances are sorted and we create a new line by fitting 25% of the closest points. The algorithm used is a simpler *least squares* version (algorithm 5.1), as the *total least squares* implementation from assignment 4 was not working correctly. If the line hasn't changed since the last execution, it is randomly altered by a not too large amount. The distances from the line to the points are recomputed and the step is repeated. The iteration of this step stops when a predefined number of points, 5% in this exercise, is very close to the line.

3. Optional step. Take the closest points to line (in this case, the points that are less than 5 pixels away from the line) and try to fit a new line, but this time using a least median squares regression, to work deal with outliers.

4. The points that are closest to the fitted line are removed as they are considered as belonging to the line and step 1 is repeated to fit the next line.

5. At the end we have all the lines (4) and count how many pixels they contain together. If the number of pixels that are not contained by any line is larger than 1% the entire algorithm is repeated.

Steps 1-5 are repeated a small number of times and at each step the solution is compared to the best solution found in the previous iterations. A solution is better if less pixels are left out. If a 0 pixel loss solution is found the algorithm makes an early exit. If after the a number of iterations there is no solution with less than 1% percent pixel loss, we conclude that there is no solution and return the best solution as a best effort result, but not as a good one. With this procedure, the rest of the regions are eliminated: if a region contour cannot be approximated to 4 lines then it is probably not a ARToolKit marker.

Algorithm 5.1 Python code describing the algorithm used for computing *least squares fitting*, used instead of the incorrectly implemented total *least least squares* from the previous assignment. Code found at [2].

```
from math import *
n=input("Enter number of points:")
sum_x=0
sum_y=0
sum_xx=0
sum_xy=0
for i in range(1,n+1):
    print "For point %s"%i
    x=input("Enter x:")
    y=input("Enter y:")
    sum_x=sum_x+x
    sum_y=sum_y+y
    xx=pow(x,2)
    sum_xx=sum_xx+xx
    xy=x*y
    sum_xy=sum_xy+xy
#Calculating the coefficients a=(-sum_x*sum_xy+sum_xx*sum_y)/(n*sum_xx-sum_x*sum_x)
b=(-sum_x*sum_y+n*sum_xy)/(n*sum_xx-sum_x*sum_x)
print "The required straight line is Y=%sX+(%s)"%(b,a)
```

5.2 Results

On images without noise the results are good, but still in some cases there exists an obvious error. One such case is for the image 5.1 used in the previous part of the chapter, as can be seen in images 5.5 and 5.6.

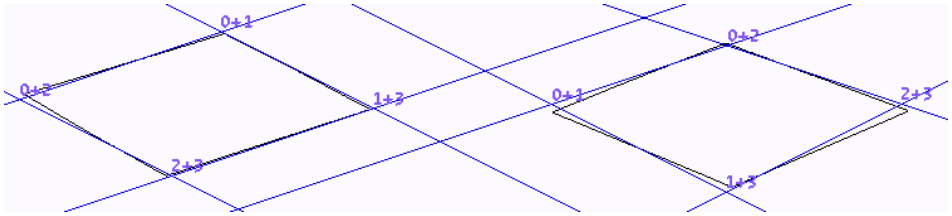

Figure 5.5: The contours, the found line and intersection points for image 5.1. There are visible problems at intersection $0+2$ of the first contour, while for the second contour, only intersection $0+2$ is accurate. Running the algorithm more times does give a better solution, but also takes longer.

Running the algorithm more times will bring better results, as can be seen in image 5.7. Also, running step 3 of the algorithm will improve the results even more, as presented in image

5.2.1 Adding noise

5.2.1.1 HSV noise

In image 5.9 the behaviour of the algorithm on a not to noisy image can be seen. The noise was applied with *The GIMP,* using the menu `Filters->Noise->HSV noise`. The corner 2+3 on the second marker has a big error, without a visual reason as to what influenced it (the contour can be seen in figure 5.10). The reason for this is that outliers remain from fitting the first lines, so line number 3 tried to fit also all the points that did not belong the first lines. Finally the fit was obtained from randomly changing the computed line (as described in step 2 of the algorithm). Increasing the tolerance of the line (maximum distance to consider the line containing a point) destabilises the first lines, but after a larger number of runs a good result can finally be found.

Obviously, the high contrast in this particular image makes the task manageable.

5.2.1.2 Hurl noise

The test image was obtained using GIMP with the menu Filter->Noise->Hurl. The high contrast again allows for good results, and, but only if the threshold for considering a point fitted is increased for the $5 pixels$ default, to, for example, $10\,pixels$. In figure 5.11 the results for threshold $5\,pixels$ can be seen. *Line* 2 is wrong for both markers, while the reasonable expectation would be to have

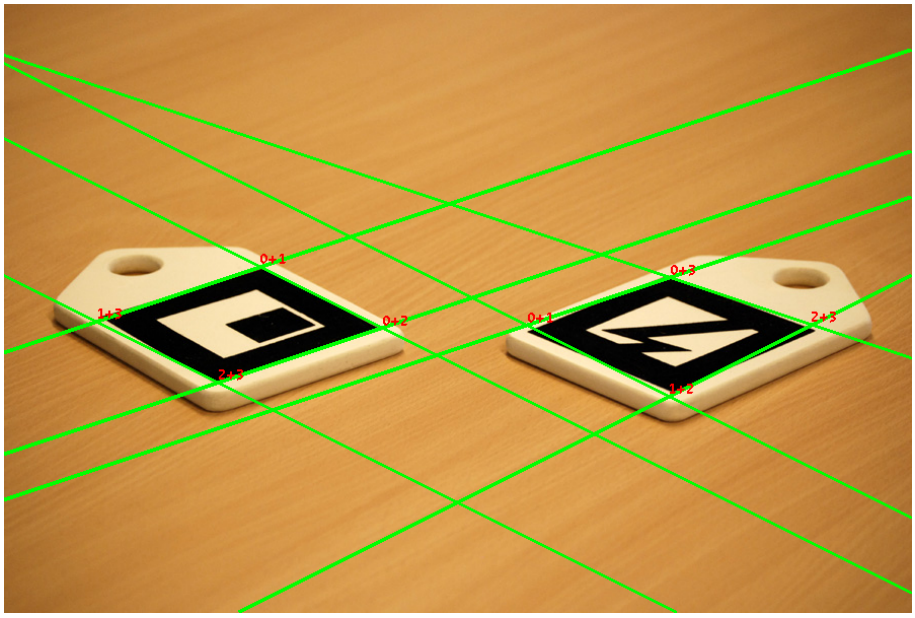

Figure 5.6: The lines and points found, drawn on image 5.1. The number of runs is limited to 40 and step 3 is not executed.

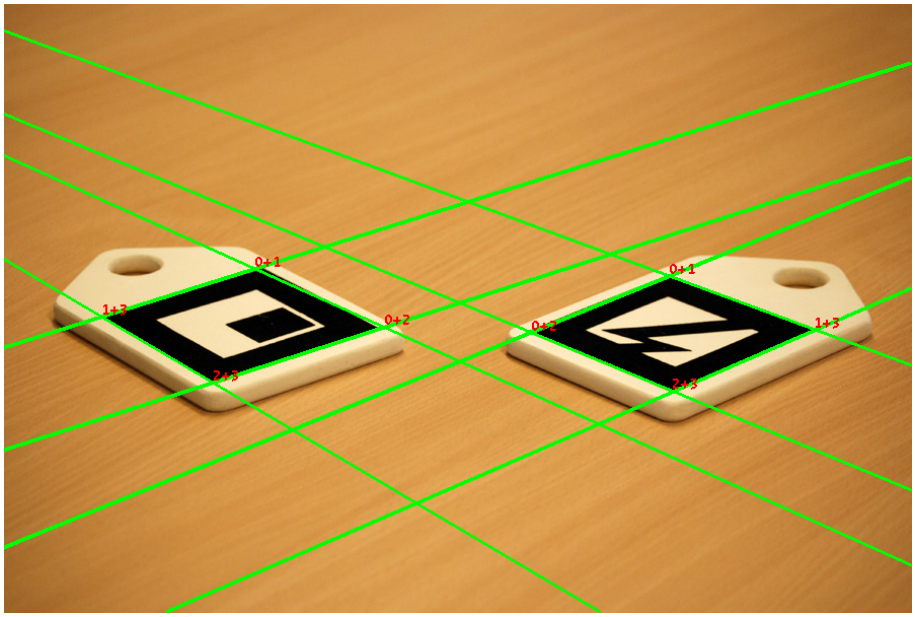

Figure 5.7: Running the algorithm more times will in the end produce better results.

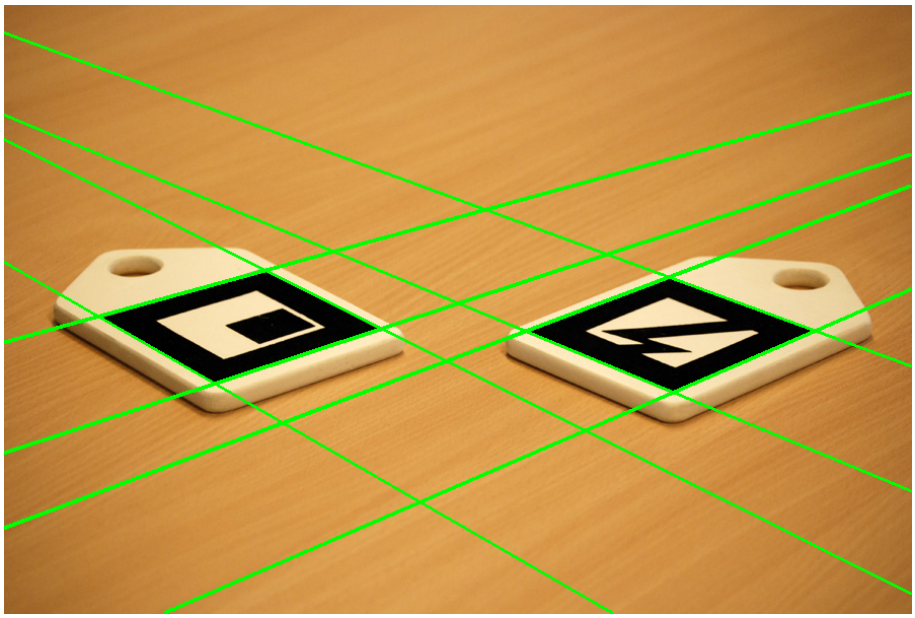

Figure 5.8: The precision of the line is improved running a least medians squares fitting after approximating with least squares, as described in step 3.

this error for the last computed line, *line* 3. The first thought was that the code putting the numbers on the image is wrong, but this was not the case. The brief investigation performed did not show a reason for this behaviour. The threshold of 10 *pixels* solved this image too (image 5.13).

Again, the high contrast insured clear contour points and the success of the procedure.

5.2.1.3 Conclusion on adding noise

As long as the contours can still be constructed the line fitting algorithm will work. Running a hurl filter from the previous subsection with 2 steps instead of one already makes the contour not break and there's nothing more to do for the incremental line fitting. For example, increasing the Randomisation parameter from 50 to 59 for the imge 5.11 already make the contours dissapear.

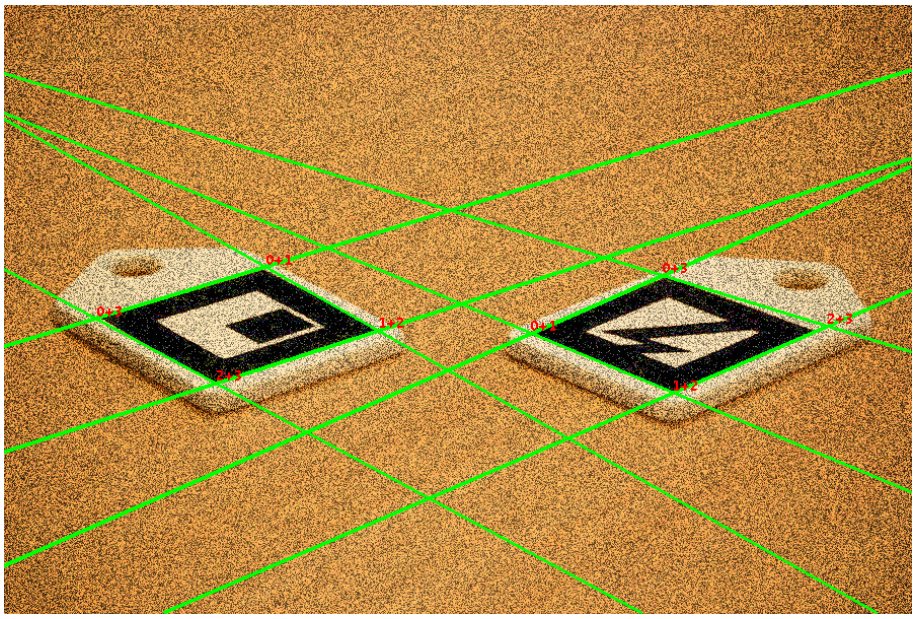

Figure 5.9: Noisy image, obtained applying a HSV noise filter in GIMP, with the following parameters: $Holdness = 1$, $Hue = 8$, $Saturation = 10$ and $Value = 138$.

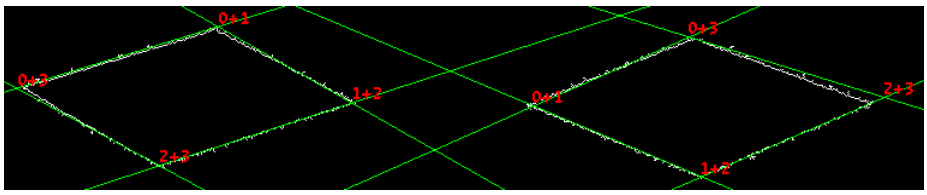

Figure 5.10: Contour and lines of the image 5.9. There is no obvious reason for why corner $2 + 3$ has such a big error. The line 3 was actually obtained by applying random modifications to the one obtained by least square, so a slightly random line was the best result.

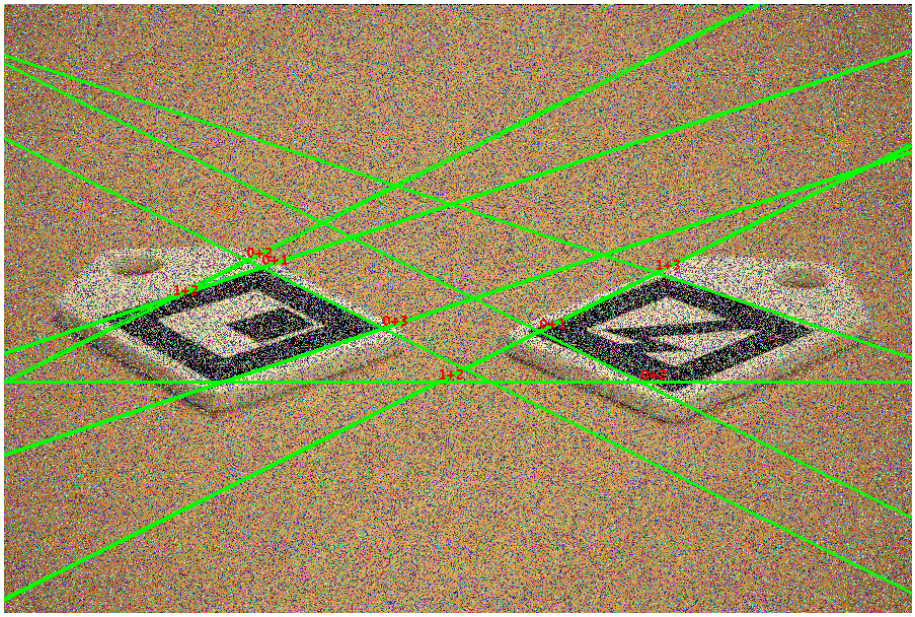

Figure 5.11: Hurl noise applied to the original image, using GIMP, with parameters *Randomseed* = 10, *Randomization* = 50 and *Repeat* = 1. With the default distance threshold (5 *pixels*) for considering a point as contained by a line the results very wrong. Line number 2 is wrong in both markers, cases while the other ones are OK. After a brief investigation I could not explain why it was *line* 2 that was wrong and not *line* 3.

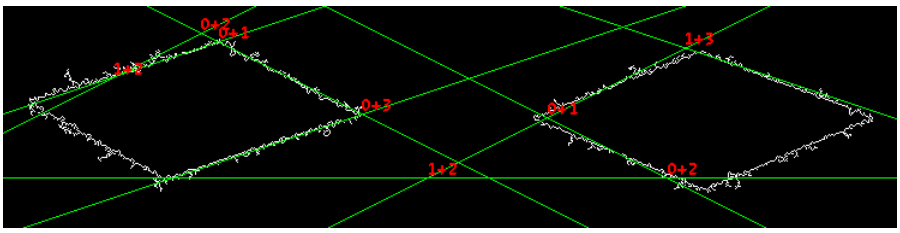

Figure 5.12: Contour and found lines for image 5.11.

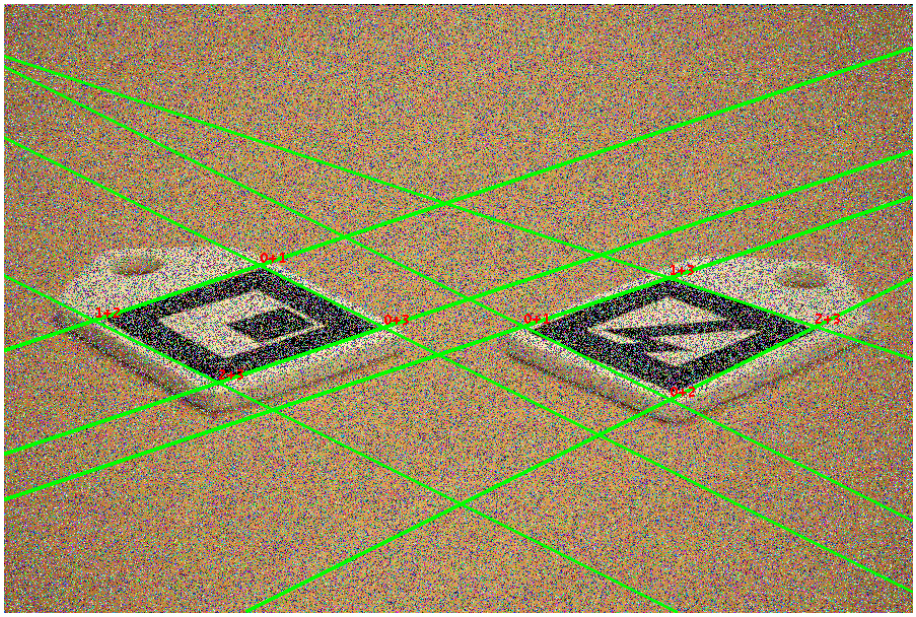

Figure 5.13: Increasing the threshold for considering a points fitted solved the problem from image 5.11.

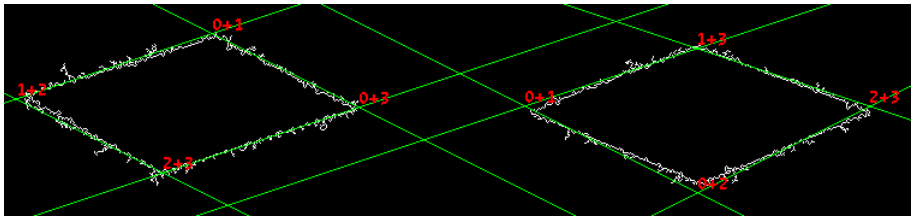

Figure 5.14: Contour and lines for image 5.13.

Chapter 6

Texture Segmentation

6.1 Law's Texture Energy Measures

Law's Texture Energy Measures are a mean of characterising texture properties based on *texture energy* measurements. Energy measurement is done by applying a set of 16 two-dimensional filters obtained by combining pairs of one-dimensional filters and finally obtaining a 9 element feature vector that describes each pixel[21].

6.1.1 Texture filters

4 one-dimensional filters are combined in pairs to obtain the 16 two-dimensional filters, they are:

$$\begin{aligned} Level : H_L &= [\ 1 \quad 4 \quad \mathbf{6} \quad 4 \quad 1\ ] \\ Edge : H_E &= [\ -1 \quad -2 \quad \mathbf{0} \quad 2 \quad 1\ ] \\ Spot : H_S &= [\ -1 \quad 0 \quad \mathbf{2} \quad 0 \quad -1\ ] \\ Ripple : H_R &= [\ 1 \quad -4 \quad \mathbf{6} \quad -4 \quad 1\ ], \end{aligned} \tag{6.1}$$

with the values in bold indicating the origin in each filter and their name indicating the type of texture events they should measure (H_L-Gaussian, local average; H_E-gradient, responds to edges; H_S-LOG, detect spots; H_R-Gabor, detect ripples[4]). The two-dimensional filters are created by taking the outer products of all the combinations between two one-dimensional filters, done as

in the following example[21]:

$$H_{EL} = H_E^T \cdot H_L = \begin{bmatrix} -1 \\ -2 \\ -0 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{6.2}$$
$$= \begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}.$$

These filters can then be applied as convolution kernels to the target image. The same result is obtained by doing the convolution in two steps, one for each of the one-dimensional the filters in the pair, a method which is more efficient computationally.

Out of the total 16 combinations, the symmetric pairs of maps are combined, thus leaving 9 final maps[23, 21]:

$$\begin{array}{cc} \frac{1}{2}(H_{LE} + H_{EL}) & \frac{1}{2}(H_{LS} + H_{SL}) \\ \frac{1}{2}(H_{LR} + H_{RL}) & H_{EE} \\ \frac{1}{2}(H_{ES} + H_{SE}) & \frac{1}{2}(H_{ER} + H_{RE}) \\ H_{SS} & \frac{1}{2}(H_{SR} + H_{RS}) \\ H_{RR} & \end{array} \tag{6.3}$$

6.2 K-Means Clustering

The k-means algorithm is an algorithm for *clustering* n objects into k partitions, $k < n$, based on their properties, with the number k known in advance. It's basic idea is to first cluster the points randomly or heuristically in k partitions, then, repeatedly, define the partitions' properties (*centroid* or *mean point*) based on their new member objects and cluster the objects again. The fact that k needs to be known in advance is one of the algorithm's big drawbacks.[13]

6.2.1 K-means clustering implementation description

The k-means implementation used in the assignment is actually the most common form of the k-means algorithm, named *Lloyd's algorithm*, according to Wikipedia page on k-means[13].

Initialisation step. The algorithm starts by choosing k points in the image, called centres, and compute the centroid based on the features of that point. These centres are chosen by spreading them uniformly in the image and then moving them by on a short distance in a random direction. After the first step the centres are recomputed only to have a mark in the case

the texture are organised in only k regions (this was indeed the case for most test images), but they will not play any special role in centroid computation again.

Refinement loop. Each of the points in the images is a assigned a new cluster, the one which has the most similar properties. The similarity is computed as a distance between the feature vectors, in the implementation this was the ordinary Euclidean distance; a simple sum of absolute differences was also tried, the results where almost the same, even better in some cases, but nothing clearly provable.
The centroids are then recomputed, simply taking the average of the features of the points assigned.
The refinement loop repeats until the number of pixels that change their partition remains relatively constant and below a threshold that is proportional with the total number of pixels in the image.

6.3 Results and Implementation Aspects

The implemented ImageJ plugin did the following main steps:

1. Applying the 16 filters and obtaining an image for each of them.
2. Applying a convolution with a Gaussian filter on the 16 images, thus smoothing the images and obtaining the *energy maps*. (As a curiosity, Gaussian filtering was applied on the original image, before steps 1, with comparable results.)
3. Combining the 16 energy maps into the final 9 energy maps, which will give the feature vector for each of the pixels.
4. Running a k-means algorithm to partition the pixels.

The results where highly dependent on the following main factors:

- Scaling of the texture.
- The choice of the original centroid.
- The radius of the Gaussian kernel convolution applied at step 2. Small value performed badly, the suggested value of $15x15$ and slightly above (ex. $20x20$, $25x25$) gave the best results, over $30x30$ the results where again worse.

A final try was to add surrounding regions of uniform colour to the texture images, but this did not improve the results, but actually gave strong edges which would form their own cluster.

6.3.1 Energy maps

The energy maps for the H_E filter combinations, all but H_{EE}, where visually useful for recognising edges, but did not seem to a homogeneous colour for the region (this may seem obvious, because their actual purpose is to recognise edges, but remember that a Gaussian was applied to spread the value). All in all, the H_{RS}property seemed to give different intensities to different regions. These are exemplified in figure6.2, which shows the intensities in the maps computed for image 6.1.

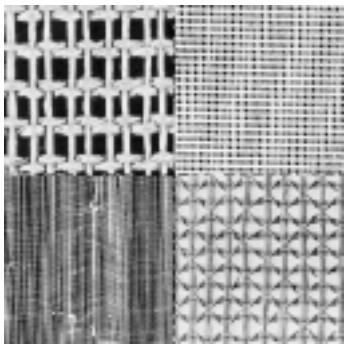

Figure 6.1: 4 textures from the assignment image, scaled by 1/2.

6.3.2 Segmentation results

6.3.2.1 The standard 16 textures image from the assignment document

The following four images show the 16 textures from the assignment document, and the results of applying the four steps described at the beginning of the section, step 2 using a Gaussian filter of dimensions $15x15$ and the starting centre points evenly distributed with a grain of randomness. The centres are also computed shown, as there are as many regions are different textures , but given the quality of the result they don't have a meaning.

6.3.2.2 Fewer textures

Segmenting images with fewer partitions gave slightly better results. Reference image is figure6.7. The same steps as in previous subsection are applied.

6.3.2.3 Applying Gaussian filtering on the images

As an experiment, the Gaussian filtering was applied on the image before the map creation. Given that the quality of the previous results was low, the result of this experiment were on par with the standard technique. The best result

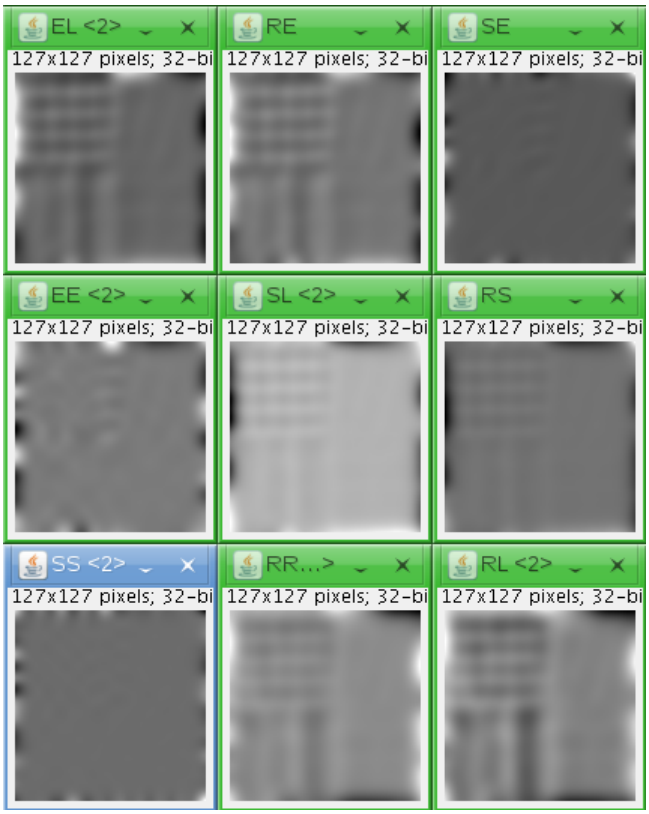

Figure 6.2: The 9 energy maps for the image in figure 6.1.

is presented in figure 6.10. By chance, this is one of the best results from the entire exercise.

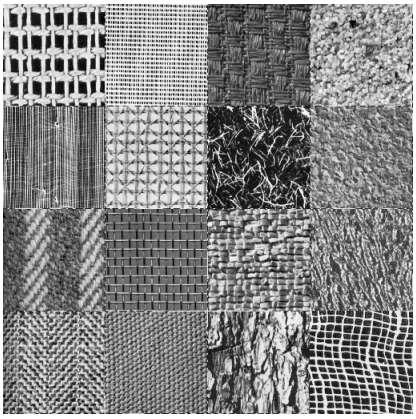

Figure 6.3: The 16 textures test image from the assignment[17].

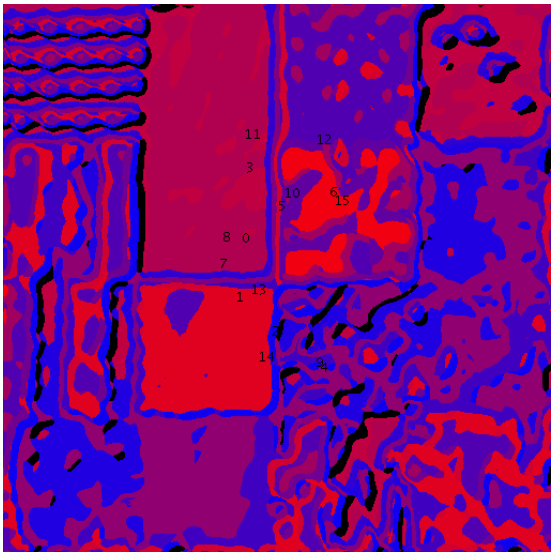

Figure 6.4: Texture segmentation on the original 16 textures image (figure 6.3), standard 4 steps, Gauss filter $15x15$, starting centres evenly distributed.

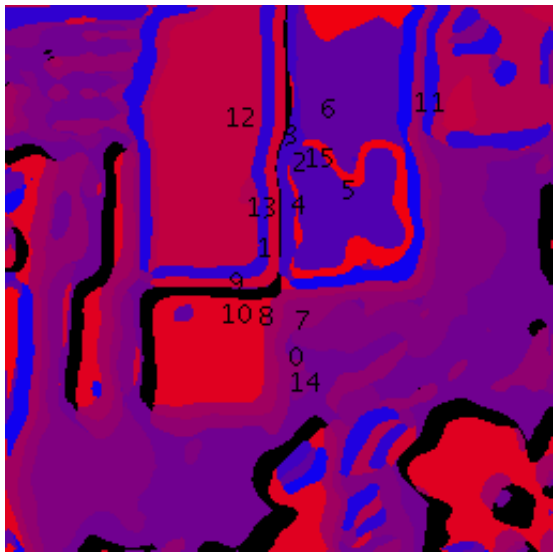

Figure 6.5: Texture segmentation on image 6.3, image scaled by 1/2.

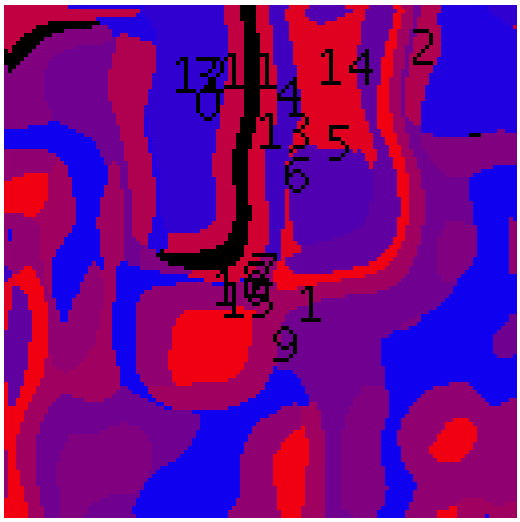

Figure 6.6: Texture segmentation on image 6.3, image scaled by 1/4.

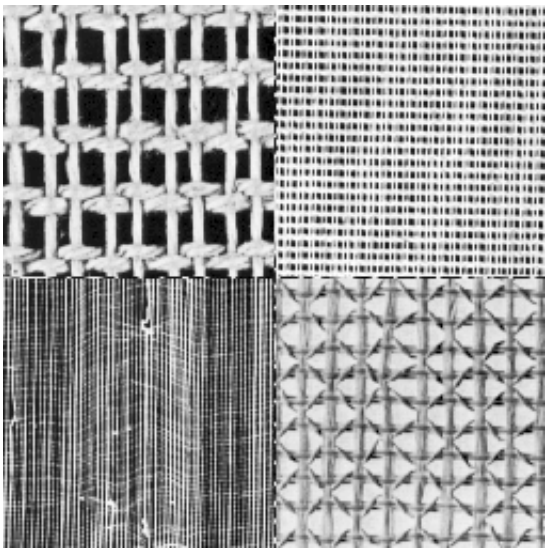

Figure 6.7: First 4 textures from image 6.3.

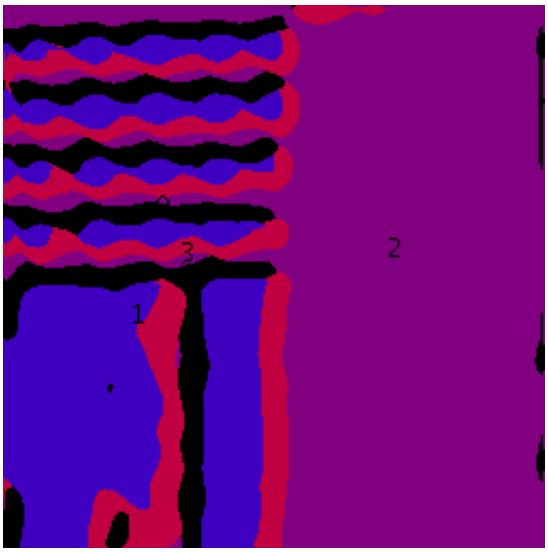

Figure 6.8: Texture segmentation on image from figure 6.7.

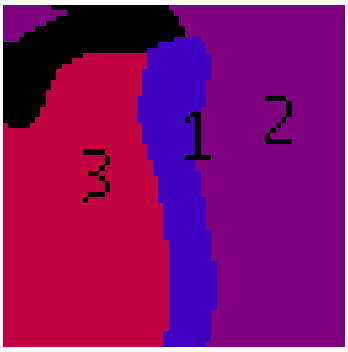

Figure 6.9: Texture segmentation on image from figure 6.7, image scaled by 1/4.

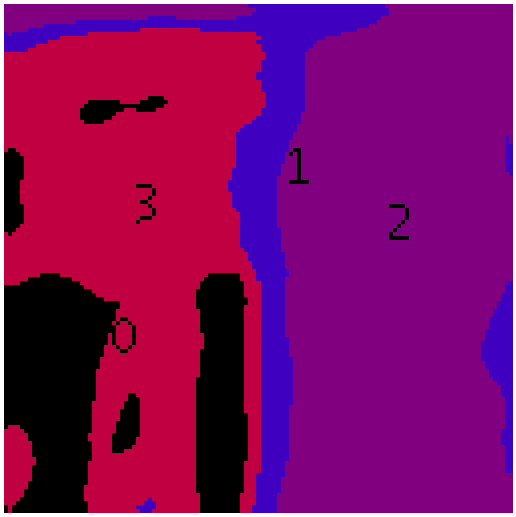

Figure 6.10: Applying the Gaussian filter on the image instead of the 16 filtered images. The image is the one from figure 6.7, scaled by 1/2.

Chapter 7

Motion Detection

7.1 Running Average on Grayscale Movies

The *running average*, also called *moving average* or *rolling average* is an a statistical measure of the average of a set of successive values. Intuitively, keeping a running average works like this: the average value is initialised, and as new values come a a small part of the old values is forgotten and small part of the new value is memorised. The new memorised part is called *learning rate* (noted α from now on) and it should b in the range [0, 1], and the forgetting rate is $1 - \alpha$. Pseudocode for implementing a running average would be:

Algorithm 7.1 Running average description.

```
runAvg := initialValue
while exist new values:
    runAvg := runAvg * (1-updateRate) + newValue * updateRate
```

In the motion detection implementation using running average, a running average value is kept for each of the pixels in the image. The running average values are initialised to the using the pixels in the first frame. With a learning rate $\alpha = 0.05$ the running average is already good enough for motion detection in many cases, as can be seen in the file **handwave-run-avg-5proc.avi** which shows motion detection done on the file **handwave.avi**. In a given frame k, for each pixel at position $(column = u, row = v)$ (denoted$I_k(u, v)$), we have a running average $B_k(u, v)$, that in the greyscale case is computed simply from the intensity value of the pixel. We then compute a difference Δ_k between B_k and the pixel; if Δ_k is larger than a an established threshold (denoted $\tau_k(u, v)$) we consider that there has been a movement in the image. A good threshold for the test images was 3 (the intensity range is $[0, 255]$).

7.2 Running Gaussian Average on Greyscale Movies

The *running Gaussian average* is an average measure that tries to keep an account of the variability of the values. Thus, in our motion detection case for each pixel (u, v) we have two measures: the *mean value* $\mu(u, v)$ and a *spread* $\sigma(u, v)$ (the spread is a measure of *variability* [1]). The mean is computed just like the running average, while for the spread we use the formula[20]:

$$\sigma_k^2(u, v) \leftarrow \alpha \cdot (I_k(u, v) - \mu_k(u, v))^2 + (1 - \alpha) \cdot \sigma_{k-1}^2(u, v). \tag{7.1}$$

The threshold in this case computed as a multiple of the spread:

$$\tau_k(u, v) = s \cdot \sigma_k(u, v)$$

where s is a constant scale factor.

The results in using this technique where varying with the different test movies and zones in the movies. A problem with using this method is that the spread may remain at a low value because in most frames the variability is low, but there are still enough frames in which the intensity variation is big. When these somewhat bigger and rare intensity variations happen, motion is detected although it should not. Setting a higher value for the constant s would affect the regions of the image that are less noisy and motion would be missed. This effect can be seen as pulsating pixels in the video **biped-cornell-Gauss-run-5proc.avi**, made with $\alpha = 0.05, s = 3$. The reference movie is **biped-cornell.avi** and the same movie with running average is **bipep-cornell-run-avg-5proc.avi**.

7.3 Adding Background Selectivity to the Running Average methods

Adding background selectivity means taking into consideration if a pixel is in the foreground or not when updating the average values. This means: there exists a measure name if a pixel is considered to be in the foreground it's average values (*running average*, or *mean* and *spread* in the case of Gaussian average) are not updated. The changes to the code are minimal, they only need to accommodate a new decision. There are improvements when using this method, and can be seen in file **bipep-cornell-run-avg-5proc-bgmodel.avi**, where background selectivity have been added to the running average method. The original movie is the same **biped-cornell.avi** and for comparison use the movie without background selectivity **bipep-cornell-run-avg-5proc.avi**. The first difference is the smaller number of noise pixels.

[1] More at Wikipedia page on statistical variability: http://en.wikipedia.org/wiki/Statistical_dispersion

7.4 Running Averages on Colour Videos

In order to do motion detection on colour videos a measure for the distance between two colours is needed. The method used was doing a simple Euclidean distance on the 3 RGB values of each pixel. The results are mostly the same as with the greyscale movies, with a regression on a noisy movie where pixels near the horizon had the colour varying slightly between blue and green. While this wasn't a big difference visually, the pixels kept staying in the foreground.

Chapter 8

Multiple Feature Tracking

In a succession of frames multiple points can be identified, but there remains the problem of the correspondence between them. If we are watching a object containing a point $p_{1,k}$in a frame k with m points, and in frame $k+1$ we have the set of points

$$P_{k+1} = (p_{1,k+1}, ..., p_{m,k+1}),$$

we want to know which of the points from P_{k+1} is the same with the point $p_{1,k}$. Making the correspondence between all point's positions means finding the trajectory of the point. This correspondence problem is solved knowing the following properties of trajectories (these properties generally hold, but it is not a rule):

- The instantaneous velocity of an object does not change abruptly.
- The direction of motion does not change abruptly.

One algorithm for solving this problem is the Greedy Exchange Algorithm by Sethi and Jain (1987)[20].

8.1 Greedy Exchange Algorithm

8.1.1 Path coherence measures

The Greedy Exchange Algorithm defines a *path coherence* function and then attempts to improve (minimise) it's value by assigning the points in successive frames to different paths. A path is identified by it's first point (the point from the first frame). The function for measuring path coherence was defined in such a way as to respect the following principles:

1. The function should not be negative.
2. The function should consider the amount of deviation, but not the direction of the deviation.

3. The function should respond equally to increases and decreases in speed.
4. The function should be zero if the is no change in the motion[24].

A path or *trajectory* is defined on a sequence of N frames as follows:

$$T_i = [p_{i,1}, p_{i,2}, ..., p_{i,N}]$$

The values that are computed in order to measure the path coherence:

1. The *displacement vector* is a measure of the distance between two points $p_{i,k-1}, p_{i,k}$ in different frames:

$$d_{i,k} = p_{i,k} - p_{i,k-1} = \begin{pmatrix} x_{i,k} - x_{i,k-1} \\ y_{i,k} - y_{i,k-1} \end{pmatrix},$$

   where x and y are the coordinates of the points.

2. The actual coherence function:

$$\phi(d_1, d_2) = \omega_{dir} \cdot \phi_{dir}(d_1, d_2) + \omega_{vel} \cdot \phi_{vel}(d_1, d_2),$$

   where ω represent weights, ω_{dir} weight for the *direction coherence* and ω_{vel} weight for the *velocity coherence*.

3. Direction coherence:

$$\phi(d_1, d_2) = 1 - cos(\theta_{1,2}) = 1 - \frac{d_1 \cdot d_2}{\parallel d_1 \parallel \cdot \parallel d_2 \parallel},$$

   $d_1 \cdot d_2$ denoting the inner product.

4. Velocity coherence:

$$\phi_{vel}(d_1, d_2) = 1 - 2 \cdot \frac{\sqrt{\parallel d_1 \parallel \cdot \parallel d_2 \parallel}}{\parallel d_1 \parallel + \parallel d_2 \parallel}.$$

Given all the measures the coherence of the trajectory is the sum of the coherence values for all the frames. The final solution of tracking the points is found out by minimising the sum of all trajectory coherence values (do note that this is actually a measure of incoherence, the bigger the values, the more abrupt the changes in direction or velocity).[20]

8.1.2 Minimising the path coherence function value

Performing this minimisation is the actual greedy exchange algorithm.

1. **Initialisation.** The algorithm begins by assigning the points to the trajectories based on the indexes the points have in each frame. The first step that is actually recommended and was implemented is initialising the trajectories on by finding the nearest neighbour in two consecutive frames

[24], but do note that the number of distance computations is in the order of magnitude $N \cdot m^2$ (N-number of frames, and m-number of points). And this is only initialisation. The paper [24] confirms that the complexity is Nm^2 in section VI.

2. **Exchange loop.** Starting from frame 2 compute the gain of exchanging each combination of two points from of one trajectory to the other, and perform the exchange that brings the biggest gain if there exists an exchange that brings a gain. Set an exchange flag if and exchange was performed.

3. **Termination.** Check the exchange flag, is an exchange was done the algorithms goes to step 2, otherwise it ends.[24]

This algorithm can be optimised by running the exchange loop in two directions, first in increasing order of frames, and then in decreasing order of frames (the first frame continues to stay untouched, as it does not make sense to perform exchange on it). One problem that may appear is in the stopping of the algorithm if there are points that will keep being exchanged. This is solved by keeping a vector of exchanges. If the number of exchanges is constant over a maximum number of loops L (this limit is proportional to the number of frames and points, $L = N \cdot m$). The number of steps performed by the two way version of greedy exchange always stopped faster than the one way version.

One last note can be made on the similarity between greedy exchange and bubble sort. Just like bubble sort, in greedy exchange local exchanges that seem to move closer to the optimum are done. Of course, bubble sort is guaranteed to reach the optimal solution, while greedy exchange not. The next similarity is in the stopping condition, we stop when no more useful swaps are done. The last similarity is in the two way optimisation. Running in two directions makes bubble search much faster (but still slow for a sorting), just like it happens with greedy exchange.

8.2 Results

The results of running the algorithm where good, but it has to be noted that already after the initialisation the trajectories where already left in a pretty sorted out state (image 8.3). The following tests where done on scrambled trajectories (as presented in image 8.2), obtained from the reference trajectories in image 8.1. On all the examples, running the two-way version only made the algorithm finish faster, but the results were at most as good as for the one-way version.

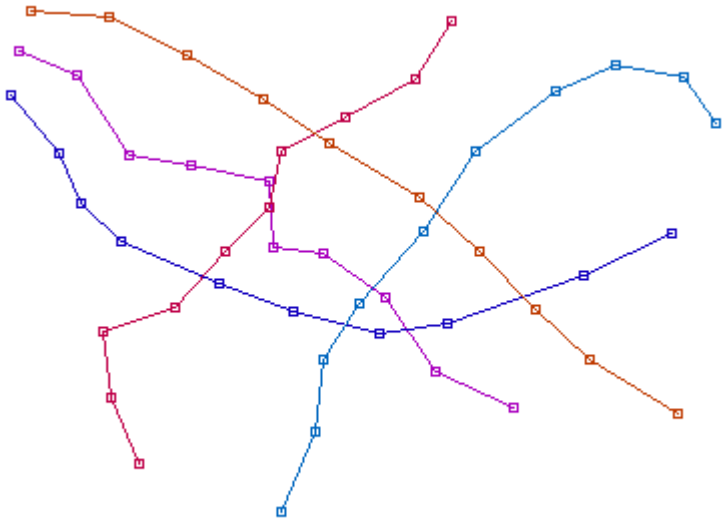

Figure 8.1: Original trajectories.

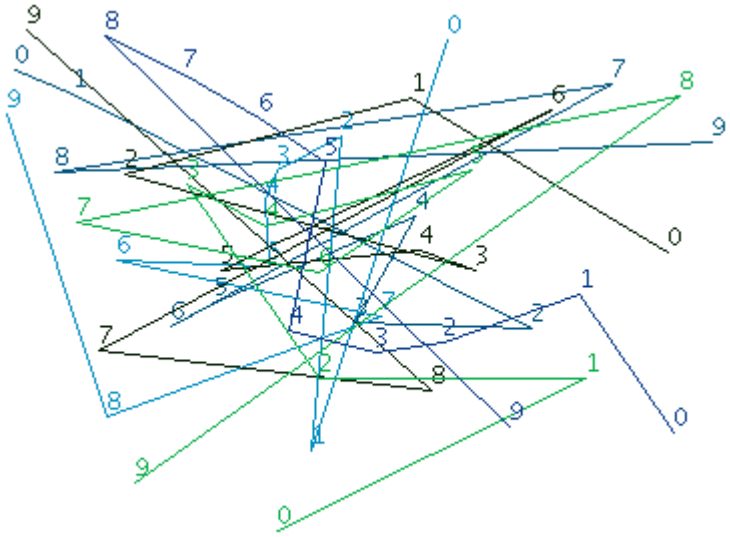

Figure 8.2: Scrambled trajectories for testing.

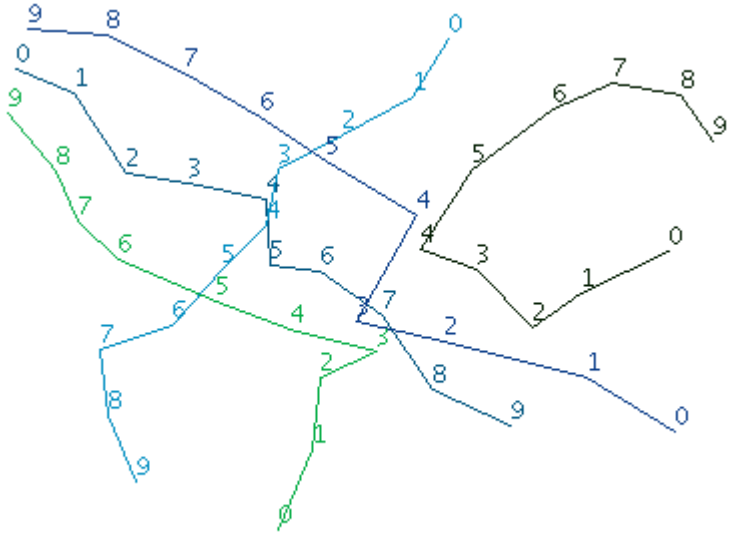

Figure 8.3: The result of the initialisation, notice that it is already pretty close to the goal.

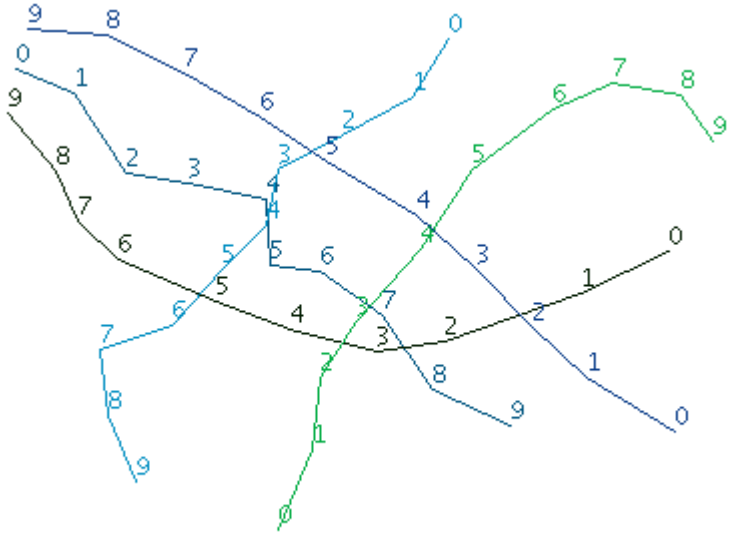

Figure 8.4: After performing simple greedy exchange. The measure of incoherence is 2.75, and the number of iterations 66.

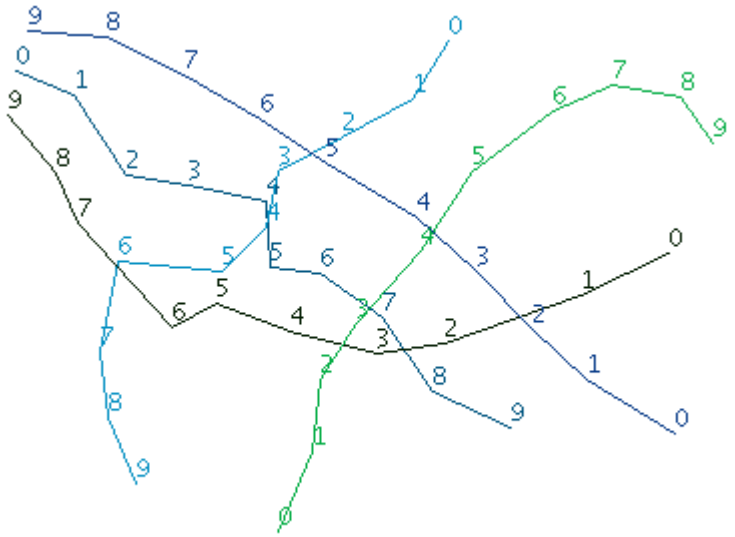

Figure 8.5: After performing greedy exchange in both directions. In this particular case the result are actually worse than with the basic variant. The measure of incoherence is 3.56.

Chapter 9

Camera Calibration

Because real cameras do not have the ideal properties of the *pinhole camera model*, a set of parameters are computed to describe the distortion introduced by the camera sensor, these are called *intrinsic camera parameters*. Another set of parameters are computed to describe the distortion introduced by the lenses - *radial lens distortion*. Finally, another set of parameters describe the coordinate system transformation from world coordinates to camera coordinates, they are called *extrinsic camera parameters*, but they are common to both the pinhole camera and real cameras, and do not measure imperfections of real camera[6, 19, 1]. All these parameters are *geometric camera parameters*.

Camera calibration (or *camera resectioning*) is the process of finding the *geometric camera parameters* of a camera that produced a given photograph or video[6].

One tool for finding camera parameters is EasyCalib[1]. Given a known planar pattern and at least two different pictures of the pattern, EasyCalib will output the following parameters[19]:

$$
\begin{array}{ccccc}
\alpha & \gamma & \beta & c_x & c_y \quad -intrinsic\ camera\ parameters \\
k_1 & k_2 & & & \quad -parameters\ for\ lens\ distortion \\
 & & & & \\
r_{11} & r_{12} & r_{13} & & \quad -rotation\ matrix\ for\ image\ 1 \\
r_{21} & r_{22} & r_{23} & & \\
r_{31} & r_{32} & r_{33} & & \\
t_x & t_y & t_z & & \quad -translation\ matrix\ for\ image\ 1 \\
 & & & & \\
r_{11} & r_{12} & r_{13} & & \quad -rotation\ matrix\ for\ image\ 2 \\
r_{21} & r_{22} & r_{23} & & \\
r_{31} & r_{32} & r_{33} & & \\
t_x & t_y & t_z & & \quad -translation\ matrix\ for\ image\ 1 \\
\ldots & & & &
\end{array}
$$

In the studied example, the output was:

[1] http://research.microsoft.com/en-us/um/people/zhang/Calib/

904.001 1.00652 906.531 280.352 219.286

0 0

0.98997 -0.0295796 0.138144
0.0160703 0.995066 0.0979017
-0.140358 -0.0946998 0.985562
-3.47623 3.45164 14.2232

0.995969 -0.000242683 0.0897001
0.0187344 0.978506 -0.205367
-0.0877223 0.206219 0.974566
-3.34225 3.55572 14.6437

0.90757 -0.0385882 0.418125
-0.00197965 0.995364 0.0961578
-0.419897 -0.0880977 0.903286
-2.54375 3.5588 15.7231

9.1 Finding the Focal Length from the Output of EasyCalib

Given the output of EasyCalib for the test image we want to find the *focal length* of the camera.

The following data is known:
$R_x = 640pix$ (the Ox resolution, known from the dimension of the image)
$R_y = 480pix$ (the Oy resolution, known from the dimension of the image)
$D_x = 6mm$ (the width of the sensor).
$\alpha = 904.001pix$
$\beta = 906.531pix$
$s_x = s_y = 1$(the sensor scales, assumed to be 1).
From here on: $\alpha = f \cdot s_x \longrightarrow f = \alpha \longrightarrow f = 904.001pix$. By proportionality:

$$f = \frac{904.001\,pix}{640\,pix} * 6mm = 8.475mm,$$

a reasonable value for a small camera. The value can be checked on the Oy axis, the other way around:

$$\frac{8.475mm}{\frac{906.631pix}{480pix}} = 4.487mm,$$

the vertical size of the sensor, close to the expected value $\frac{480}{640} \cdot 6 = 4.5mm$.

9.2 Checking the rotation matrices

A rotation matrix should be an orthogonal matrix[14], this means that $Q^TQ = QQ^T = I$.

Example, for the matrix:

$$Q = \begin{bmatrix} 0.98997 & -0.0295796 & 0.138144 \\ 0.0160703 & 0.995066 & 0.0979017 \\ -0.140358 & -0.0946998 & 0.985562 \end{bmatrix}$$

we have:

$$Q \cdot Q^T = \begin{bmatrix} 9.99999318e-01 & -6.91780000e-09 & 4.49872080e-07 \\ -6.91780000e-09 & 9.99999342e-01 & 4.89012000e-08 \\ 4.49872080e-07 & 4.89012000e-08 & 1.00000088e+00 \end{bmatrix}$$

In order to measure the difference between this matrix and the identity matrix, we define the measure *squared differences matrix*, a matrix that at element e_{ij} contains the squared difference between corresponding elements in the first two matrices. We compute this measure with the following code (Python):

Algorithm 9.1 Python code for computing the squared differences matrix

```
def SDMatrix(m1, m2, rows, cols):
    sdm = zeros([rows,cols], float)
    for i in range(0, rows):
        for j in range(0, cols):
            sdm[i, j] = pow((m1[i,j] - m2[i,j]), 2)
    return sdm
```

For all tests we obtain a matrix very close to the zero matrix. Finally we sum all the elements. For the presented example this sum was $2.0751436056e-12$, a very small number.

9.3 Compute the Projections of the Original Model onto the Image

Chapter 10

Summary and Comments

Overall the course is one of the best I have taken, and there are very few suggestions to be made. Some mathematical aspects are a bit heavy, but I can't suggest spending more on them during the lecture, in cases where big parts of the mathematical background are lacking, not much can be done in 20 minutes.

Writting this document proved to be a much bigger task than expected, and it wasn't very clear where the equilibrium is: when I should insist more in giving details and when the details where enough.

Bibliography

[1] Geometric camera parameters article at university of nevada. 9

[2] Least squares fitting algorithm in python at http://www.dreamincode.net/. 5.1

[3] Moments in image processing, article at national taiwan normal university website. 2.1

[4] Presentation on texture, at michigan state university. 6.1.1

[5] Wikipedia - artoolkit. 5

[6] Wikipedia - camera calibration. 9

[7] Wikipedia - central moment. 2.1

[8] Wikipedia - centroid. 2.1

[9] Wikipedia - coefficient of variation. 2.3

[10] Wikipedia - convex hull algorithms. 1.2.1

[11] Wikipedia - graham scan. 4

[12] Wikipedia - image moments. 2.2

[13] Wikipedia - k-means algorithm. 6.2, 6.2.1

[14] Wikipedia - rotation matrix. 9.2

[15] Wilhelm BURGER. Assignment 2, autumn 2008, computer vision course, October 2008. 2.1, 2.2

[16] Wilhelm BURGER. Assignment 4, autumn 2008, computer vision course, 2008. 4.1, 4

[17] Wilhelm BURGER. Assignment 6, autumn 2008, computer vision course, 2008. 6.3

[18] Wilhelm BURGER. Slide set 4, computer vision, 2008. 4.2.1

[19] Wilhelm BURGER. Suplementary course notes on 3d basics, 2008. 9

[20] Wilhelm BURGER. Suplementary course notes on motion analysis, 2008. 7.2, 8, 8.1.1

[21] Wilhelm BURGER. Suplementary course notes on texture, November 2008. 6.1, 6.1.1, 6.1.1

[22] Wilhelm BURGER and Mark J. BURGE. *Digital Image Processing - An Algorithmic Introduction using Java*. Springer, 2008. 1.1.1, 2.1, 5.1.2

[23] Anke Meyer-Bäse. *Pattern Recognition for Medical Imaging, Google Books*. Academic Press, 2004. Preview available at google books. 6.1.1

[24] Ishwar K. Sethi and Ramesh Jain. Finding trajectories of feature points in a monocular image sequence. *IEEE Trans. Pattern Analysis and Machine Intelligence*, January 1987. 4, 1, 3