Circle Detection Using Hough Transforms Documentation

COMS30121 - Image Processing and Computer Vision

Jaroslav Borovička – pinus@centrum.cz, jb2383@bris.ac.uk

Contents

You can find complete sets of images documenting the detection process in `images.tar.gz`. This document was written using TEX. Please direct any questions or comments to my e-mail. Last modified 04/03/2003, Jaroslav Borovicka.

1 Problem statement

The objective of this application is the recognition of circular shapes in an image. This task can be subdivided into following procedures. First, a the image is *smothed* in order to reduce the amount of noise. Then, an *edge recogition procedure* is implemented. I use the proposed first-derivative *Sobel filters* to determine the edges and edge directions. After some testing, I also added a *second-derivative* filter which, in connection with the Sobel filters, offers a more robust way of edge recognition. Then, a *circular Hough transform* is accomplished on the thresholded edge map. The resulting (a, b)-space is then convoluted with a *"Mexican hat" filter* in order to enhance the 'hot spots'. The (a, b)-space is then thresholded, and centres of the hot spots found. We then *accumulate in r-space* in order to find the most observable circle with the given centre. Setting a suitable threshold allows for *concentric circles*.

1.1 Convolution

Since only filters of small sizes are used, I programmed the convolution functional in spatial coordinates, rather than using the Fourier transform (i.e. I do $h = f * g$ directly instead of $h = \mathcal{F}^{-1}(\mathcal{F}f \cdot \mathcal{F}g)$. Although the multiplication of the Fourier images is much faster than the convolution in the spatial coordinates, the Fourier transform, implemented as Fast Fourier Transform with its complexity $O(n \log n)$, will be inefficient for convolution of the image and a small filter. The convolution is implemented in the `convolution` function, and it generally uses the (discrete) formula

$$h[k, l] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i, j]\, g[k - i, l - j]$$

but in practice, i and j only run through the support of $f[i, j]\, g[k-i, l-j]$. We also should not forget that convolution distorts the image along the edges (since it assumes a periodic function, which is not the case here, and we rather pad the image with zeros). Therefore, we should rather neglect the edges when working with the image after convolution.

1.2 Smoothing the image

Smoothing the image is accomplished using a 2-dimensional Gaussian smoothing filter. The size of the filter (standard deviation of the conjugate normal distribution) can be specified using the `-s` command-line parameter. Since approximating the values of the distribution function of normal distribution is rather complicated, I use the probability density function instead. So, since the centre of the filter is shifted to $(0, 0)$, we have

$$\text{smoothfilter}\,[i, j] = \frac{1}{(\sqrt{2\pi}\sigma)^2} \cdot e^{-\frac{i^2+j^2}{2\sigma^2}}$$

instead of (mathematically correct)

$$\text{smoothfilter}\,[i, j] = \int_{i-0.5}^{i+0.5} \int_{j-0.5}^{j+0.5} \frac{1}{(\sqrt{2\pi}\sigma)^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}\, \mathrm{d}y\, \mathrm{d}x$$

However, the error is less than 5% for a vast majority of cases, a number more than sufficient for our application. The filter values are then normalised so that they add up to one. The smoothing is then done by convoluting the above defined filter with the image (see function `smoothimage` for more detail on the application).

1.3 Application of Sobel filters

The horizontal and vertical Sobel filters are applied on the smoothed image. The filter masks are shown in Figure 1.

-1	0	1	-1	-2	-1
-2	0	2	0	0	0
-1	0	1	1	2	1

Figure 1: Vertical (left) and horizontal (right) Sobel filter masks

These filters are convoluted with the image, and then merged using the L_2 (Euclidean) norm

$$|\nabla f| = \text{sobel}_{merged}[i,j] = \sqrt{(\text{sobel}_{vert}[i,j])^2 + (\text{sobel}_{horiz}[i,j])^2} = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}.$$

sobel_{merged} then contains the edge map where each pixel contains the intensity of the edge. This edge map is then normalised and thresholded using the `-t1` parameter (values 0..255).

1.4 Implementation of a 2nd-derivative edge filter for improved edge detection

The sobel filters themselves are quite often mislead by noise which they falsely recognise as edges. This can be to a large extent solved by smoothing the image but then the edges tend to be thicker. A good solution is to support the Sobel filter by a 2nd-derivative filter. Whereas Sobel filters are based on looking-up the maxima of the first derivative (middle graph of Figure 2, the 2nd-derivative filter is looks-up zero crossings of the 2nd-derivative (right image of Figure 2).

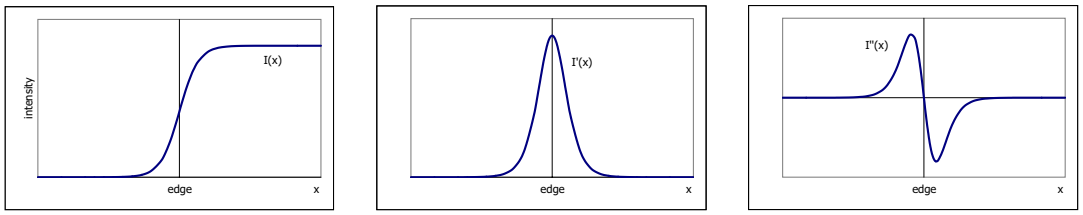


Figure 2: Behaviour of the intensity function (left), its derivative (middle), and second derivative (right) around an edge. The first derivative reaches its maximum, the second derivative crosses zero at the edge.

The 2nd-derivative filter is implemented in the following way. To reduce the occurrence of high frequencies (noise), we smooth the image, and then apply the Laplace operator on the smooth image, i.e.

$$\nabla^2(G * f) = \frac{\partial^2(G * f)}{\partial x^2} + \frac{\partial^2(G * f)}{\partial y^2}$$

where f is our image, and G is the Gaussian smoothing filter. From the linearity of convolution, it follows that

$$\nabla^2(G * f) = (\nabla^2 G) * f.$$

$(\nabla^2 G)$ (Laplacian of Gaussian – LoG) can be precomputed, and has the "Mexican hat" form (two "Mexican hat" filters of different sizes are defined in the beginning of the source code, page 12). After applying the "Mexican hat" filter, the zero-crossings have to be found. This is done by running a 2x2 window over the image — if both positive and negative values appear in the window, a zero crossing occurs. A point is then considered as an edge point when the Sobel filter signalises an edge and the "Mexican hat" filter signalises a zero-crossing. See function `edgesusing2ndder` for implementation.

1.5 Calculating the direction map

In order to calculate the direction map (see function `thresholdandfinddirectionmap`), we make use of the already calculated convolutions of the image with the horizontal and vertical Sobel filter.

For every detected edge point $[i, j]$ we calculate the angle θ as

$$\theta[i,j] = \tan^{-1}\left(\frac{\partial f}{\partial y}[i,j] \Big/ \frac{\partial f}{\partial x}[i,j]\right) = \tan^{-1}\frac{\text{sobel}_{vert}[i,j]}{\text{sobel}_{horiz}[i,j]}$$

Using the function `atan2` in `C` gives angles in the interval $(-\pi, \pi)$. However, since edges with angles θ and $\theta + \pi$ have got the same direction, we can shift all $\theta[i, j]$'s so that they lie in $(-\pi/2, \pi/2)$.

1.6 Accumulation into (a, b)-space using circular Hough transform

The idea of the Hough transform is that perpendiculars to edge points of a circle cross in the centre of the circle. Therefore, if we draw perpendicular lines to every edge point of our edge map, we should obtain bright 'hot spots' in the centres of the circles. We therefore draw following line segments into the (a, b) space:

$$\left.\begin{aligned} a &= r\sin\theta \\ b &= r\cos\theta \end{aligned}\right\} \text{ where } r \in (minr, maxr)$$
$$A(i \pm a, j \pm b) \leftarrow A(i \pm a, j \pm b) + E(i, j)$$

where $(minr, maxr)$ is the range of circle radii that are taken into account (customisable from command-line), A is the (a, b)-space array, and $E(i, j)$ is the strength of the edge, given by merging the two Sobel filters. A schematic picture is shown in Figure 3. This transform will create spots with

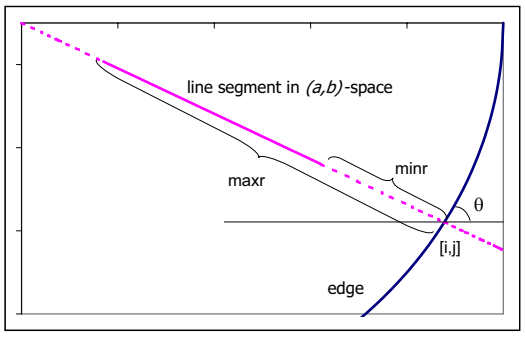


Figure 3: Calculating the line segments perpendicular to the edge to be drawn into the (a, b)-space

higher brightness in places where centres of circles should be found. However, these spots can be quite diffused and dimmed, particularly if the circles are rather distorted into ellipses (see practical results). Therefore it is necessary to concentrate these hot spots. I use convolution with a 17x17 "Mexican hat" filter which tends to concentrate the highest brightness in the centre of gravity of the hot spot. Then, the (a, b)-space is thresholded so that isolated spots are found. In these isolated spots, I find local maxima using a recursive algorithm which finds all local maxima with a minimum given distance (see function `removesmallervalues`). There can be more than one maximum in one spot if the spot is large enough. These maxima are then considered being centres of the sought-after circles.

Strictly speaking, the intensity of the line that is drawn into the (a, b)-space should decrease with factor $1/r$, i.e.

$$A(i \pm a, j \pm b) \leftarrow A(i \pm a, j \pm b) + \frac{E(i, j)}{r}$$

because without this factor, large circles, consisting of 'more points', will accumulate higher value in the centre of the circle than small circles, where less points will contribute. On the other hand, due to inaccuracies in estimating the angle of the line, lines coming from greater distance will show larger dispersion in the area of the centre, which will balance the higher number of contributing points. It is therefore ambiguous which approach should be rather used. I tested both methods, and could not say that one method would outperform the other one.

1.7 Accumulation into r-space

After the localisation of the circles' centres, the circles' radii have to be found. This is done by accumulating in a one-dimensional space which coordinate is the radius of concentric circles with the given centre. For every (discrete) radius in the desirable range $(minr, maxr)$, the sum of edge strengts $E()$ for the points P along the circle with the given radius is calculated:

$$R(r) = \sum_{P \in \text{circle}(r)} E(P)$$

Two improvements are implemented here. First, the above method would clearly prefer large circles, with large circumference. We therefore divide the accumulator by the radius in order to obtain a non-dimensional quantity. Second, we still take into account all edge points on the circle, even those whose perpendicular does not point towards the centre of the circle, and which therefore cannot form part of it. If ψ is the angle of the line connecting the centre with a given edge point, then we take into account only edge points whose perpendicular lies in $(\psi - \epsilon, \psi + \epsilon)$, where ϵ is the permitted error of the calculated edge angle. Values like $\epsilon = \pi/8$ or $\epsilon = \pi/4$ work well, reducing the amount of false edge points significantly, and still taking into account most of true edge points.

The r-space is then thresholded (using the `-t3` command-line parameter), which creates separate intervals with non-zero values. The maximum value in each interval is then the sought-after radius. Thresholding also influences the number of concentric circles found — setting the threshold to maximum will result in finding only the strongest circle, whereas a lower threshold will also permit less strong circles. Both alternatives are useful in practice.

See further parts of the report for practical results, and function `accumulateinrspace` for implementation.

1.8 Refining coordinates and radii to subpixel accuracy

During the whole process of finding the centres and radii, some inaccuracies could occur. It could be also desirable to consider a better than just pixel accuracy. Therefore, after finding the coordinates and radius, a search in the (a, b, r)-space in the neighbourhood of the found circle is executed. The size of the neighbourhood and the step size (≤ 1) can be set. We look for the highest peak of the accumulator function which sums the edge strengths alongside the circle (for edges with desired direction, see above), and divides the sum by the radius. The improvement in accuracy is significant (see practical results).

Since the calculated coordinates of the points that should contribute to the accumulator are non-integer, we interpolate between the neighbouring pixels. Assume that the calculated (real) coordinate is (x, y) and that $i \leq x < i + 1$ and $j \leq y < j + 1$, where i, j are (integer) pixel coordinates (see Figure 4), and that I is the intensity function, known for the pixels. Then

$$\begin{aligned} I(x,y) &= [(1-(x-i)) * I(i,j) + (x-i) * I(i+1,j)] * (1-(y-j)) + \\ &+ [(1-(x-i)) * I(i,j+1) + (x-i) * I(i+1,j+1)] * (y-j) \end{aligned}$$

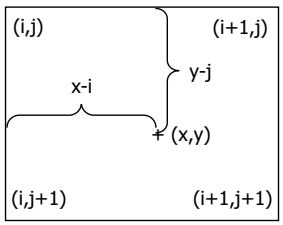


Figure 4: Schematic representation of the interpolation

2 Key issues effecting the performance

This question was already extensively dealt with in the previous section, therefore I will only summarise the main points in a table.

Problem	Solution
noisy image	smoothing the image with a Gaussian filter of adjustable size (best results for a 3×3 or 5×5 filter)
false edges, thick edges after smoothing	implementation of a 2^{nd}-derivative filter, edge points are then the zero crossings of the 2^{nd}-derivative filter which are also detected as edges by the Sobel filter
scattered 'hot spots' in the (a, b)-space	enhancement of the image using a 17 "Mexican hat" (LoG) filter
too many false hot spots in (a, b)-space	thresholding the (a, b)-space, removing hot spots with small weights ($\sum_{\Omega} I[i, j]$, where Ω is the hot spot area) after thresholding
distortions around image borders	caused by convolution, which assumes a periodic function. Remove the border areas (set the values to 0) according to the size of filters used.
several local maxima in one hot spot	using a recursive algorithm that searches the neighbourhood of the global maximum and removes lower values
accumulation into r-space contaminated by noise	we take into account only edge points whose perpendicular lies in $(\psi - \epsilon, \psi + \epsilon)$, where ϵ is the permitted error of the calculated edge angle, and ψ is the angle of the line connecting the centre with the given edge point
accumulator into r-space prefers large circles	divide values of the accumulator by the appropriate radius, i.e. $R(r) \leftarrow R(r)/r$
concentric circles	find not only the highest value in the r-space for the given centre but also other peaks that exceed the given threshold
final circles slightly inaccuracte	refinement of the coordinates and radius, searching the neigbourhood of the circle in the (a, b, r)-space for the highest accumulator value, even to subpixel accuracy, interpolation between pixels

Further comments are added in the section dealing with analysis of the results.

3 Experimental results

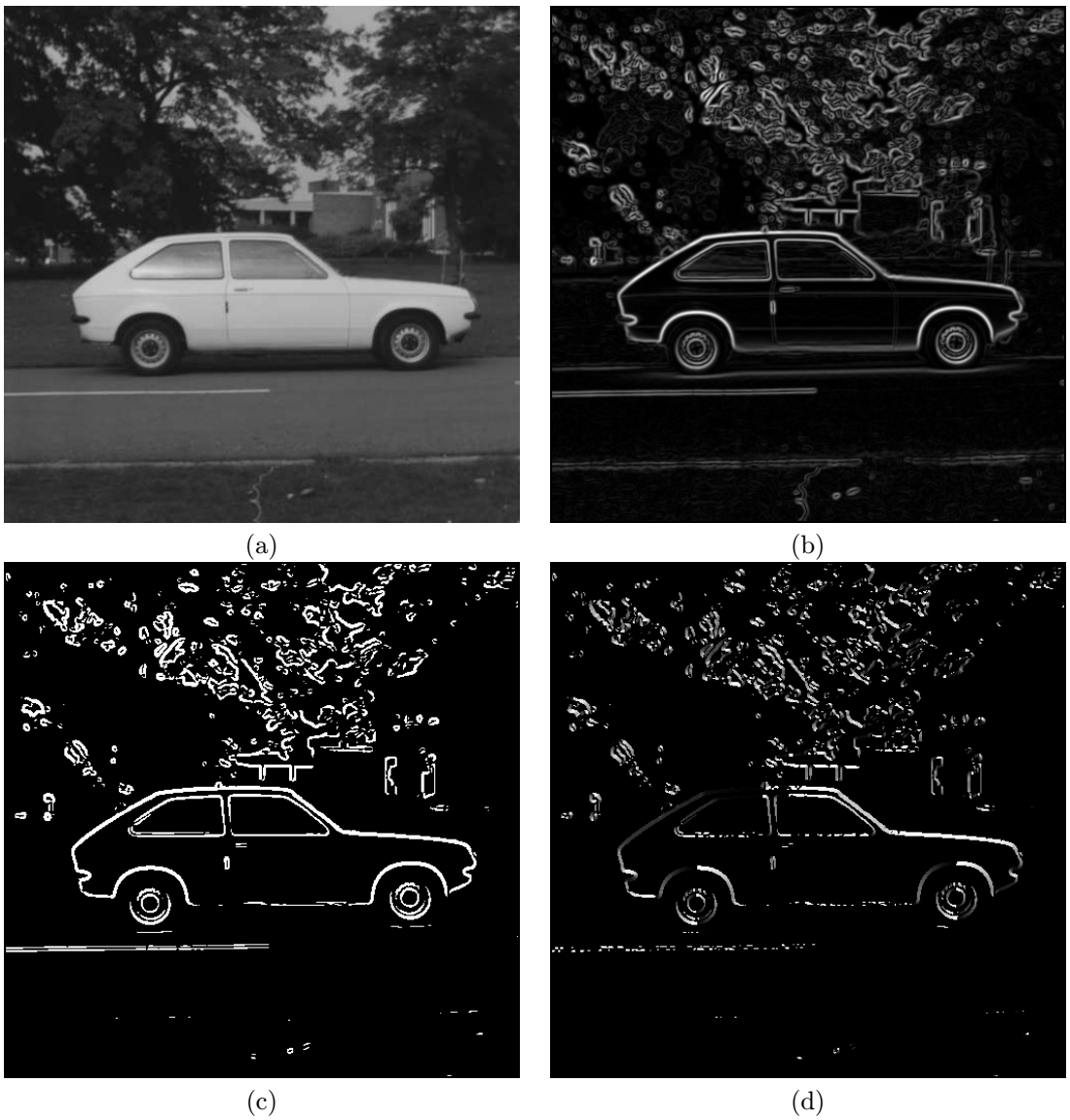
3.1 Processing `car.pgm`

First, I will concentrate on the image `car.pgm` (Figures 5, 6, 7). This picture contains lots of clutter (background tree), therefore it is desirable to set the edge threshold rather higher. However, even then the edge map (Figure 5(c)) contains a lot of edges. This can be, to a large extent, solved by using the 2^{nd}-derivative filter (Figure 6(e)). Accumulation into (a, b)-space creates two clear hot spots, strengtened by using "Mexican hat" filter (Figure 6(g)). The remaining false hot spots can be easily thresholded. When trying to find only one circle for each hot spot, the outline of the mudguard is found — allowing for concentric circles finds the most visible circles (Figure 7(i)). After refinement of the coordinates and radius, the circles give a better match, see e.g. the fact that the outline of the mudguard is not completely concentric with the remaining two circles (Figure 7(j)). This set of images was created using following parameters: edge threshold: 80 of 255, (a, b)-space threshold: 170 of 255, r-space threshold: 180 of 255, minimal radius: 3, maximal radius: 50, smoothing filter: 3×3.

3.2 Processing `coins2.pgm`

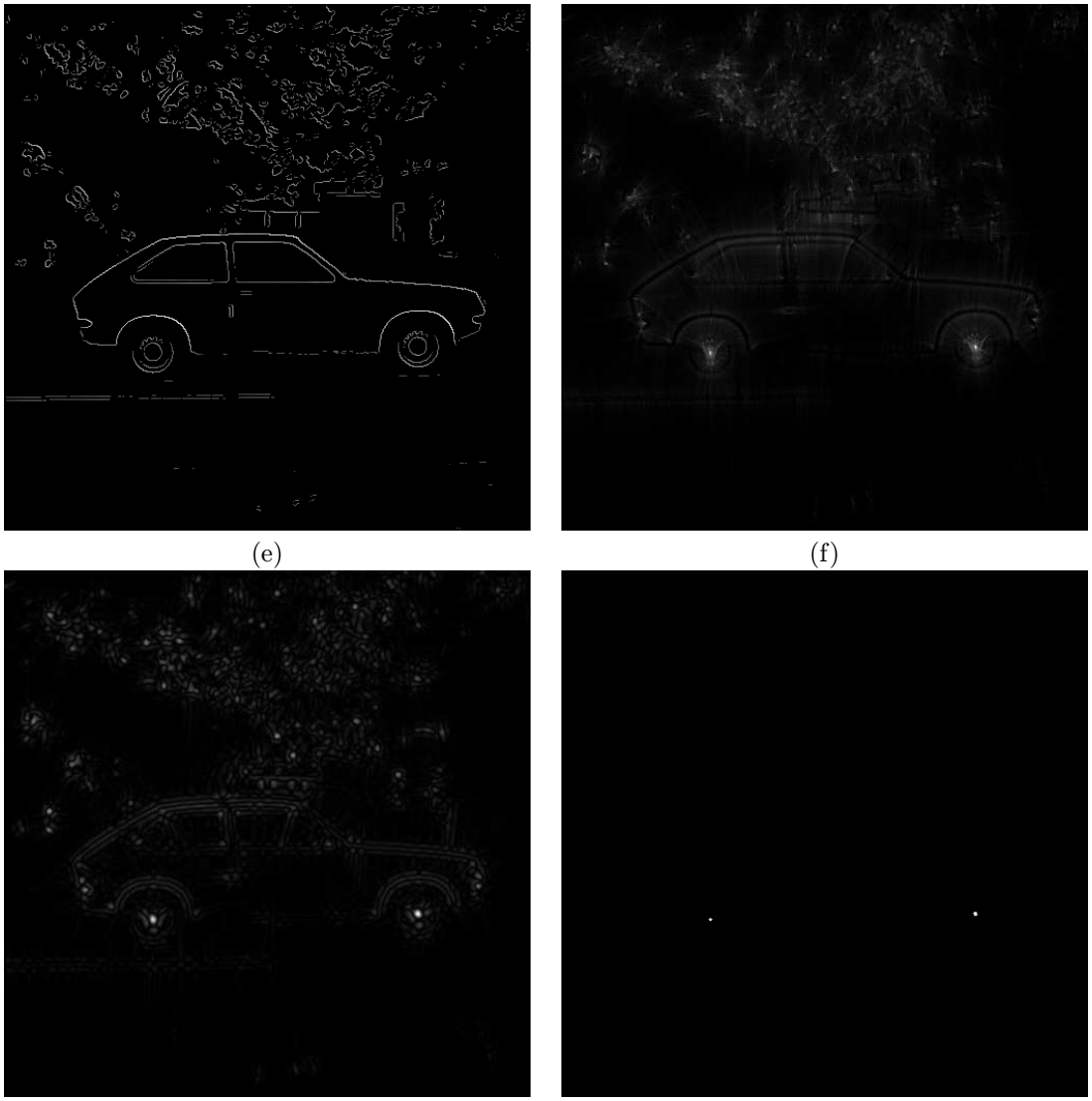
The second image I examine here in detail is `coins2.pgm`. This image also shows some peculiarities that make the circle detection difficult. Firstly, the coins in the image are quite distorted, and are rather elliptic than circular. The eccentricity is high especially for coins in the left and upper part of the image. This makes the circle fitting very difficult. Secondly, the coins overlap and some are cut by the border of the image. Again, this makes the detection uneasy, especially because border areas have to be cut because of distortions in the convolution. The most difficult coin is than the upper right one. Finally, the coins are of various brightness. Especially the left coin is quite bright, with blurred edge, and contains a lot of noisy texture (moreover, it is very elliptic and not fully visible).

All these factors require a more sensitive setting of threshold parameters than in the case of `car.pgm`. In order not to lose contour of the bright coins, the edge threshold should be set low. This transfers a lot of noise into the edge map (Figure 8(c)) — this can be partly removed by smoothing and using the 2^{nd}-derivative filter (Figure 9(e)). Again, the threshold for the (a, b)-space should be rather



(a) (b) (c) (d)

Figure 5: Processing `car.pgm` : (a) original picture (b) merged Sobel filters (c) thresholded edge map (d) direction map, mapping of angle $\theta : (-\pi/2, \pi/2) \rightarrow (0, 255)$, θ as defined in Figure 3. *Continued in Figure 6.*



(e) (f)

(g) (h)

Figure 6: Processing `car.pgm` *(continued)*: (e) edge map improved by using the 2nd derivative filter (f) accumulation into (a, b)-space (g) conentrating 'hot spots' using a "Mexican hat" filter (h) thresholded (a, b)-space. *Continued in Figure 7.*

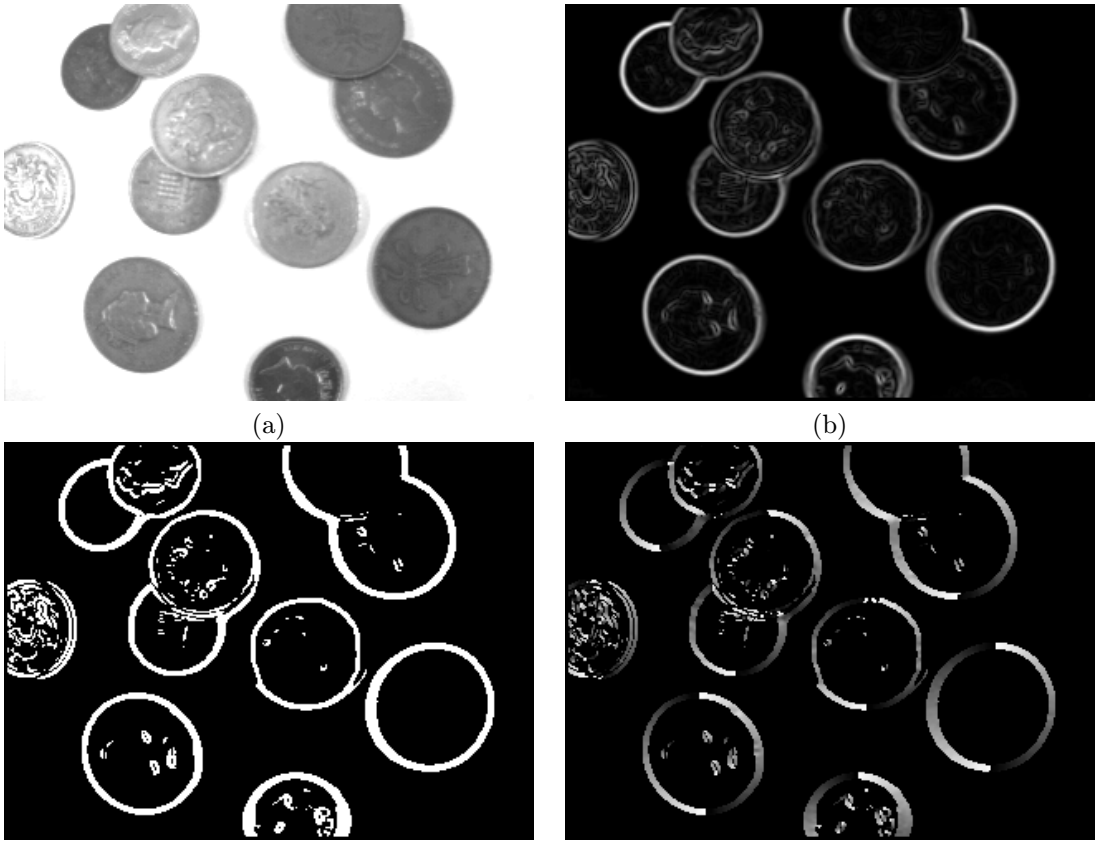
low since there will be large differences in the hot spot brightness (circular dark coins with bright hot spots, elliptical, partly visible, bright coins with less bright hot spots — see Figure 9(f)(g)(h)). However, it is difficult to set the threshold so that all correct circle centres are included and no incorrect centres are detected.

The final circle detection produces some slightly inaccurate circles, so that it is desirable to do a final refinement. From the 11 coins, detection of 7 of them is improved, 2 remain the same, and detection of 2 is slightly worsened. This worsening in detection is caused by the ellipticity - the refinement finds the highest value of the accumulator, this however occurs when a part of the coin is matched perfectly, whereas the match in another part of the coin is worsened. This can shift the detected circle away from the subjectively best position which however does not match any part of the coin edge perfectly (i.e. large coin bottom left). Note also the improvement for the left coin and the partly visible top right coin).

The set of images was created using following parameters: edge threshold: 50 of 255, (a, b)-space threshold: 60 of 255, r-space threshold: 255 of 255, minimal radius: 10, maximal radius: 50, smoothing filter: 3×3.

(i) (j)

Figure 7: Processing `car.pgm` *(continued)*: (i) detected circles (j) detected circles after subpixel refinement



(a) (b)

(c) (d)

Figure 8: Processing `coins2.pgm` : (a) original picture (b) merged Sobel filters (c) thresholded edge map (d) direction map, mapping of angle θ : $(-\pi/2, \pi/2) \rightarrow (0, 255)$, θ as defined in Figure 3. *Continued in Figure 9.*

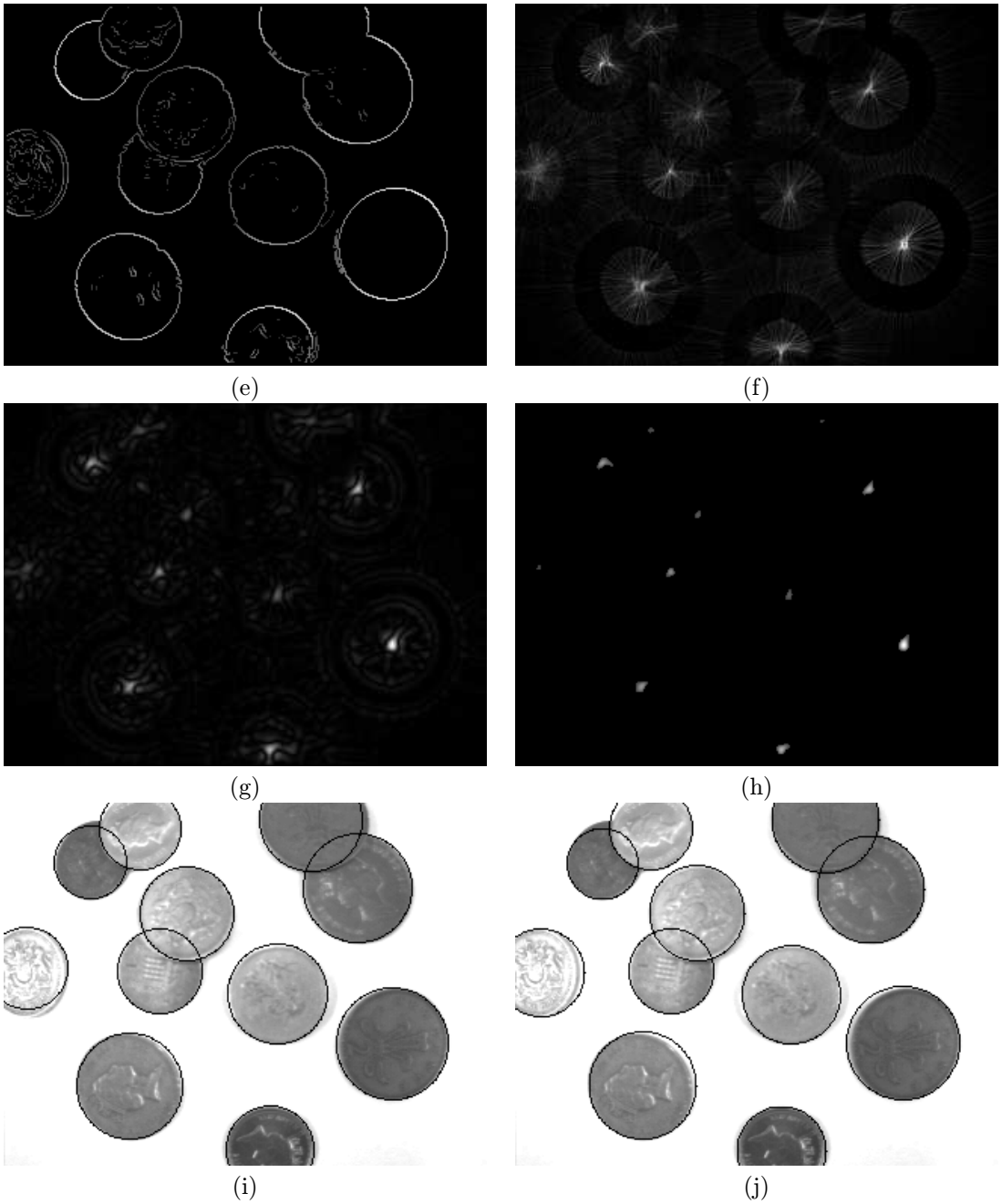
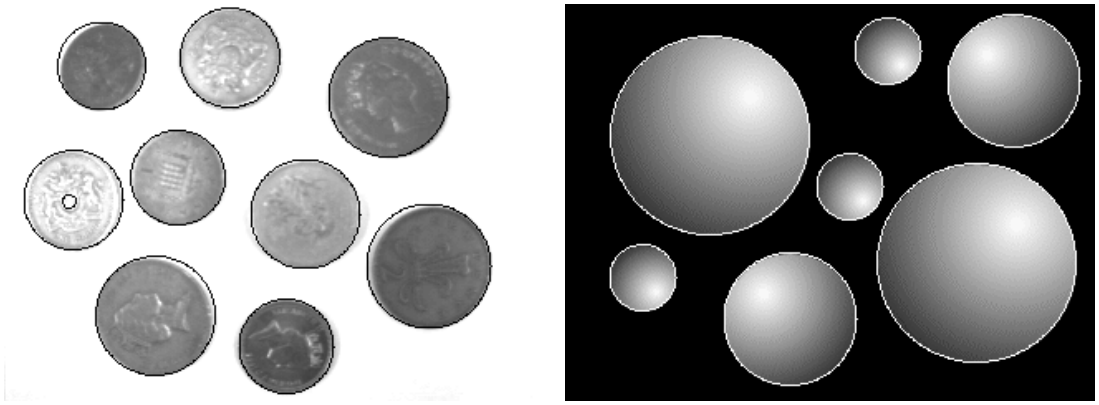


Figure 9: Processing `coins2.pgm` *(continued)*: (e) edge map improved by using the 2^{nd}-derivative filter (f) accumulation into (a, b)-space (g) conentrating 'hot spots' using a "Mexican hat" filter (h) thresholded (a, b)-space (i) detected circles (j) detected circles after subpixel refinement

3.3 Remaining images

The image `coins1.pgm` is very similar to `coins2.pgm` but less difficult. Note again worse match for the elliptic coins, and the ability to detect the middle hole in the left coin. The image `spheres1.pgm` is very easy. Therefore I only show the final results, after refinement (Figure 10). You can find complete sets in the file `images.tar.gz`.



(a) (b)

Figure 10: (a) Detected circles in `coins1.pgm` (b) Detected circles in `spheres1.pgm`

4 Analysis of the results, possible improvements

As seen from previous images, the circle detector works quite well. However, there are still problems remaining. The image `spheres1.pgm` is very easy to deal with. The image `car.pgm` is more difficult but the hot spots corresponding to the wheel centres are still far brighter than the false hot spots. The only tricky thing is to set the r-space threshold to find all the concentric circles correctly. The image `coins1.pgm` and especially `coins2.pgm` are rather difficult. It is not simple to find a suitable combination of thresholds which would let all correct circles pass, and remove all false circles. Then it is necessary to use suitable smoothing and apply the 2^{nd}-derivative edge detector to make finding the thresholds easier. Following summary explains this in more detail.

The main problem is the sensitivity of the detector to correct setting of the thresholding parameters. Often it happened that either too many or too few circles were detected.

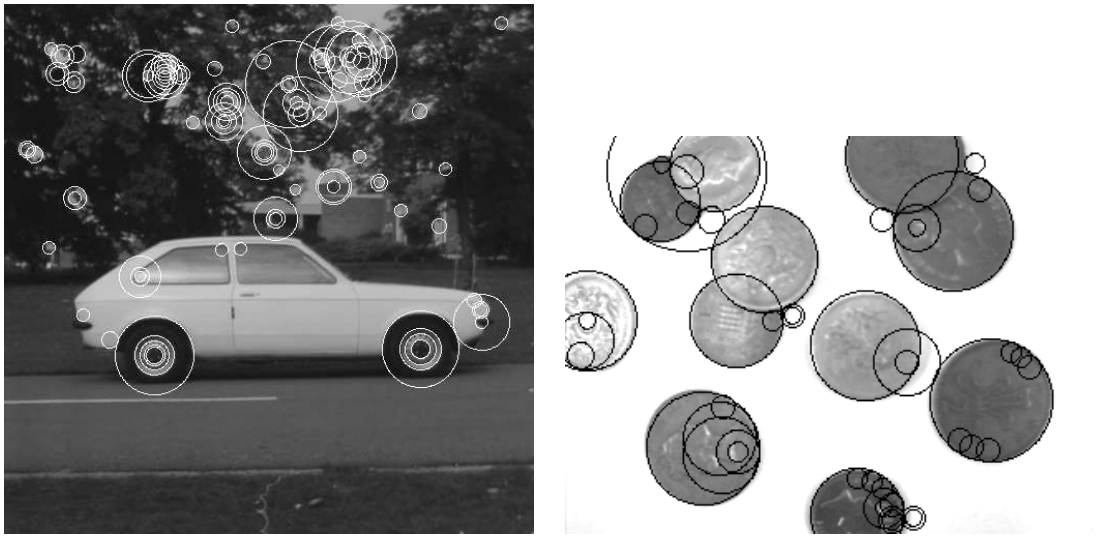
This problem was reduced by the implementation of image pre-smoothing and by using the 2^{nd}-derivative edge detector together with the Sobel filters. However, thresholding parameters should still differ for various types of images. We use three thresholds: thresholding the edge map, thresholding the (a, b)-space, and thresholding the r-space.

For noisy images, a significant amount of noise can be reduced by smoothing (i.e. in `coins1.pgm`, most of the noise coming from the 'textures' of the coins can be removed in this way). In such a case, the threshold of the edge map can be set quite low in order to keep as many edges as possible. However, if the image is strongly cluttered (tree in background of `car.pgm`), smoothing does not help, and we should set the edge threshold higher.

The (a, b)-space threshold influences how many hot spots will be recognised as centres. If we are pretty sure that the centres will show similar concentration of intersecting lines (wheels in `car.pgm`) we can set the threshold high, keeping only the unambiguous centres. On the other hand, if the circles have various edge strenghts, or some of them are distorted into ellipses, a lower threshold should be set, in order to preserve also the less visible hot spots. This can however cause problems since also false centres might pass such a lower threshold (Figure 11).

The r-space threshold only influences the extent to which concentric circles are permitted. Setting it to maximum value (255) finds only the most visible circle, setting the threshold lower allows for concentric circles.

The above problems can be significantly simplified if the number of circle centres is known. In this case, thresholding the (a, b)-space, which is the most difficult of all, becomes obsolete — we simply take the n brightest spots, where n is the number of circles to be expected. We can expect this e.g. in the case of recognising the wheels of a car. The two hot spots, corresponding to the wheel centres, are by far brightest in the image. If we know there are only two circles in the image, the detection in `car.pgm` becomes extremely easy. The same applies for concentric circles if we know



(a) (b)

Figure 11: Consequence of setting the (a, b)-space threshold too low, together with permitting very small circle radii. (a)`car.pgm` (b)`coins2.pgm`

the maximum number of concentric circles.

Similarly, a significant simplification is the known range of circle radii. It is especially important to set the lower bound as far as possible — as we can see in Figure 11, circles with small radii often occur as noise. Since we often try to detect objects of similar radii (coins), this assumption can often be made.

It is also helpful to know the minimum distance between circle centres, since then, after finding one hot spot, we can wipe out all values in the (a, b)-space which are closer than the given distance.

If none of above assumptions can be made, we could still run the detection program several times and keep the centres that survive various parameter combinations.

The problem with the elliptic distortion can be solved by using elliptic Hough tranform rather than circular. We will then find a number of ellipses some of which will show only small eccentricity. These can be then declared circles.

Because of the sensitivity to the parameter setting, I would not consider this circle detector useful for general images. If we could make any assumptions about the number of expected circles (two wheels of a car) or the circle radii (radius of the coins), the detector becomes useful in practice. But we should still try to submit images where all the circles have similar contrast, and completely lie inside the image.

5 Program listing

```
#include "hough.h"

#define MAX(a,b) ((a)>(b) ? (a) : (b))
#define MIN(a,b) ((a)<(b) ? (a) : (b))
#define ABS(a) ((a)>0 ? (a) : -(a))
#define SIGN(a) ((a)>0 ? 1 : -1)

// ============================================================================
// define two "Mexican hat" filters for further use as 'hot spot' enhancer
// and second derivative filter

  float mexhatlarge[17][17] =
    {{ 0, 0, 0, 0, 0, 0,-1,-1,-1,-1,-1, 0, 0, 0, 0, 0, 0},
     { 0, 0, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0, 0, 0, 0},
     { 0, 0,-1,-1,-1,-2,-3,-3,-3,-3,-3,-2,-1,-1,-1, 0, 0},
     { 0, 0,-1,-1,-2,-3,-3,-3,-3,-3,-3,-3,-2,-1,-1, 0, 0},
     { 0,-1,-1,-2,-3,-3,-3,-2,-3,-2,-3,-3,-3,-2,-1,-1, 0},
     { 0,-1,-2,-3,-3,-3, 0, 2, 4, 2, 0,-3,-3,-3,-2,-1, 0},
     {-1,-1,-3,-3,-3, 0, 4,10,12,10, 4, 0,-3,-3,-3,-1,-1},
     {-1,-1,-3,-3,-2, 2,10,18,21,18,10, 2,-2,-3,-3,-1,-1},
     {-1,-1,-3,-3,-3, 4,12,21,24,21,12, 4,-3,-3,-3,-1,-1},
     {-1,-1,-3,-3,-2, 2,10,18,21,18,10, 2,-2,-3,-3,-1,-1},
     {-1,-1,-3,-3,-3, 0, 4,10,12,10, 4, 0,-3,-3,-3,-1,-1},
     { 0,-1,-2,-3,-3,-3, 0, 2, 4, 2, 0,-3,-3,-3,-2,-1, 0},
     { 0,-1,-1,-2,-3,-3,-3,-2,-3,-2,-3,-3,-3,-2,-1,-1, 0},
     { 0, 0,-1,-1,-2,-3,-3,-3,-3,-3,-3,-3,-2,-1,-1, 0, 0},
     { 0, 0,-1,-1,-1,-2,-3,-3,-3,-3,-3,-2,-1,-1,-1, 0, 0},
     { 0, 0, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0, 0, 0, 0},
     { 0, 0, 0, 0, 0, 0,-1,-1,-1,-1,-1, 0, 0, 0, 0, 0, 0}};

  float mexhatsmall[5][5] =
    {{ 0, 0,-1, 0, 0},
     { 0,-1,-2,-1, 0},
     {-1,-2,16,-2,-1},
     { 0,-1,-2,-1, 0},
     { 0, 0,-1, 0, 0}};

// ============================================================================
// PParams ProcessParameters(int argc, char* argv[])
//
// Processes command-line parameters, prints help.
//
// Parameters: int argc     - number of command-line parameters
//             char* argv[] - array containing command-line parameters
// Returns:    processed command line parameters
// ============================================================================

PParams ProcessParameters(int argc, char* argv[])
{
  PParams P;
  int argindex;

  // ----------------------------------------------------------------------------
  // print help

  if (argc > 1)
  {
    if ((strcmp(argv[1], "-h")== 0) || (strcmp(argv[1], "-?")== 0))
    {
       printf ("\n");
       printf ("Usage: hough inputfilename outputfilename [parameters]\n");
       printf ("       hough -h or sobel -?    get this help\n");
       printf ("Parameters:\n");
       printf ("  -t1 value    set edge threshold to 'value' (default 50)\n");
       printf ("  -t2 value    set (a,b)-space threshold to 'value' (default 50)\n");
       printf ("  -t3 value    set (r)-space threshold to 'value'\n");
       printf ("                 set this value to less than 255 in order to allow \n");
       printf ("                 for concetric circles (default 255) \n");
       printf ("  -minr value  set minimal circle radius to 'value'\n");
```

```
      printf ("  -maxr value  set maximal circle radius to 'value'\n");
      printf ("  -s value     smooth the image first, size of mask is (2*value+1)\n");
      printf ("  -d           use 2nd derivative filter to improve edge look-up\n");
      printf ("  -r           do post-calculation adjustment, and refine\n");
      printf ("                 coordinates to subpixel accuracy\n");
      printf ("  -v           save all intermediate files (incl.edge and direction map)\n");
      printf ("  -c value     set colour of the detected circles to 'value'\n");
      printf ("  -h | -?      get this help\n\n");

      exit(0);
    }
  }

  if (argc < 3)
  {
    printf ("Not enough parameters. Run 'hough -h' to get help.\n\n");
    exit(0);
  }

  // -----------------------------------------------------------------------
  // process parameters

  P = (PParams) calloc(1, sizeof(TParams));

  // this is the parameter structure
  /*
    typedef struct
    {
      int contthreshold;       // edge map threshold
      int abthreshold;         // (a,b)-space threshold
      int rthreshold;          // r-space threshold (allow for concentric circles)
      int use2ndderivative;    // should the 2nd derivative filter be used?
      int minradius;           // minimum allowed radius of the circles
      int maxradius;           // maximum allowed radius of the circles
      int smoothimage;         // size of smoothing filter
      int saveall;             // should all intermediate files be saved?
      int circlecolour;        // colour to use to draw the circles
      int refinecoordinates;   // should a final refinement of coordinates be used?
      char *inputfilename;
      char *outputfilename;
    } TParams;
  */
  // default values
  P->minradius = 5; P->maxradius = 50;
  P->contthreshold = 50; P->abthreshold = 50; P->rthreshold = 255;
  P->smoothimage = 0;
  P->saveall = 0;
  P->refinecoordinates = 0;
  P->circlecolour = 255;

  strcpy (P->inputfilename = (char *) malloc(strlen(argv[1])+1), argv[1]);
  strcpy (P->outputfilename = (char *) malloc(strlen(argv[2])+1), argv[2]);

  argindex = 3;

  // loop for parameter processing
  while (argindex < argc)
  {
    // threshold in the contour map
    if ((strcmp (argv[argindex],"-t1") == 0) && (argindex < argc - 1))
    {
      if (atol(argv[argindex+1]) != 0)
      {
        P->contthreshold = atol(argv[argindex+1]);
      }
    }
    // threshold in the (a,b)-space aggregation
    else if ((strcmp (argv[argindex],"-t2") == 0) && (argindex < argc - 1))
    {
      if (atol(argv[argindex+1]) != 0)
      {
        P->abthreshold = atol(argv[argindex+1]);
```

```
      }
    }
    // threshold in the (r)-space aggregation
    else if ((strcmp (argv[argindex],"-t3") == 0) && (argindex < argc - 1))
    {
      if (atol(argv[argindex+1]) != 0)
      {
        P->rthreshold = atol(argv[argindex+1]);
      }
    }
    // minimal radius
    else if ((strcmp (argv[argindex],"-minr") == 0) && (argindex < argc - 1))
    {
      if (atol(argv[argindex+1]) != 0)
      {
        P->minradius = atol(argv[argindex+1]);
      }
    }
    // maximal radius
    else if ((strcmp (argv[argindex],"-maxr") == 0) && (argindex < argc - 1))
    {
      if (atol(argv[argindex+1]) != 0)
      {
        P->maxradius = atol(argv[argindex+1]);
      }
    }
    // should the picture be smoothed?
    else if ((strcmp (argv[argindex],"-s") == 0) && (argindex < argc - 1))
    {
      if (atol(argv[argindex+1]) != 0)
      {
        P->smoothimage = atol(argv[argindex+1]);
      }
    }
    // set the colour of the detected circles
    else if ((strcmp (argv[argindex],"-c") == 0) && (argindex < argc - 1))
    {
      P->circlecolour = atol(argv[argindex+1]);
    }
    // use 2nd derivative filter
    else if (strcmp (argv[argindex],"-d") == 0)
    {
      P->use2ndderivative = 1;
    }
    // do post-calculation adjustment and refinement
    else if (strcmp (argv[argindex],"-r") == 0)
    {
      P->refinecoordinates = 1;
    }
    // save all intermediate files
    else if (strcmp (argv[argindex],"-v") == 0)
    {
      P->saveall = 1;
    }
    argindex = argindex + 1;
  }

  P->minradius = MAX(P->minradius, 1);
  P->maxradius = MAX(P->minradius, P->maxradius);

  printf ("\nParameters: (a,b)-space threshold: %d, contour threshold: %d\n",
          P->abthreshold, P->contthreshold);
  printf ("  minimal radius: %d, maximal radius: %d, smoothing %d, 2nd derivative filter: %s\n",
          P->minradius, P->maxradius, P->smoothimage, ((P->use2ndderivative)?"YES":"NO"));
  printf ("  save all images: %s, input file %s, output file %s\n\n",
          ((P->saveall)?"YES":"NO"), P->inputfilename, P->outputfilename);

  return P;
}

// ============================================================================
// void writefloatpgm (float **im, int w, int h, char *fname, float shift)
```

```
//
// converts the array of floats into an array of unsigned chars,
// normalising the values into the interval 0..255, and
// writes the array of unsigned chars into a .pgm file
//
// Parameters: float **im   - array to be converted
//             int w, int h - width and height of the array (image)
//             char *fname  - filename to be used
//             float shift  - value to be added to every pixel
// Returns:    nothing
// ============================================================================

void writefloatpgm (float **im, int w, int h, char *fname, float shift)
{
  unsigned char **imch;
  int i, j;
  float maxval;

  maxval = 0;
  imch = uchar_array (w, h);

  // find highest and smallest value (if negative values occur)
  for (i = 0; i < h; i++)
    for (j = 0; j < w; j++)
    {
      if (ABS(im[i][j] + shift) > maxval) maxval = ABS(im[i][j] + shift);
    }

  // normalise
  if (maxval > 0)
    for (i = 0; i < h; i++)
      for (j = 0; j < w; j++)
        imch[i][j] = (char)floor(ABS(im[i][j] + shift)/maxval*255);

  writepgm (imch, w, h, fname);

  free_uchar_array (imch, h);

  return;
}

// ============================================================================
// int removesmallervalues (float **f, int i, int j, int h, int w)
//
// Removes all pixels in the surrounding of the given pixel at (i,j) which
// brightness values are less then those of the pixel at (i,j).
// This method works recursively - before a pixel is removed (its value set
// to zero), this function is called again on this pixel, and first, its
// neighbourhood is examined.
// If a pixel with higher brightness occurs in the neighbourhood of the pixel
// at (i,j), the pixel at (i,j) is removed.
// The function is used for finding the maxima (hot spots) in the (a,b)-space.
//
// Parameters: float **f    - array containing the examined image
//             int i, int j - pixel which surrounding is to be examined
//             int w, int h - width and height of the array (image)
// Returns:    the final value of the pixel at (i,j)
// ============================================================================

int removesmallervalues (float **f, int i, int j, int h, int w)
{
  int k, l;
  float curval, endval;

  curval = f[i][j]; endval = f[i][j];

  // run through the neighbourhood
  for (k = -4; k < 5; k++)
  {
    for (l = -4; l < 5; l++)
    {
      // if the coordinates are in the image
```

```
      if ((i + k >= 0) && (j + l >= 0) && (i + k < h) && (j + l < w) &&
          ((k != 0) || (l != 0)))
      {
        // if the pixel has got smaller value than the pixel at (i,j)
        if (f[i + k][j + l] < curval)
        {
          if (f[i + k][j + l] > 0)
          {
            // call the function on the pixel at (i+k, j+l)
            removesmallervalues (f, i + k, j + l, h, w);
            f[i + k][j + l] = 0;
          }
        }
        else
          // there is a pixel with higher value in the neighbourhood ->
          // set the value of the pixel (i,j) to 0 at the end
          endval = 0;
      }
    }
  }
  f[i][j] = endval;
  return endval;
}

// ============================================================================
// float **convolution (float **image, int iw, int ih,
//                      float **filter, int fw, int fh)
//
// Calculates the discrete 2D-convolution of the two arrays: image and filter.
// The coordinates of the filter are centered (i.e. (0,0) in the centre of
// the filter). It is therefore adviceable (but not necessary)
// to use filters with odd height and width.
//
// Parameters: float **image  - array containing the image
//             int iw, int ih - width and height of the image
//             float **filter - array containing the filter
//             int fw, int fh - width and height of the filter
// Returns:    pointer to the convoluted array
// ============================================================================

float **convolution (float **image, int iw, int ih,
                     float **filter, int fw, int fh)
{
  float **output;
  int i, j, k, l;

  // Allocate memory for the output array
  output = float_array (iw, ih);

  for (k = 0; k < ih; k++)
  {
    for (l = 0; l < iw; l++)
    {
      output[k][l] = 0;
      for (i = MAX(k-(fh-1)/2,0); i <= MIN(k+fh/2,ih-1); i++)
      {
        for (j = MAX(l-(fw-1)/2,0); j <= MIN(l+fw/2,iw-1); j++)
        {
          output[k][l] += (image[i][j])*(filter[k-i+fh/2][l-j+fw/2]);
        }
      }

    }
  }

  return output;
}

// ============================================================================
// float **smoothimage (float **im, int w, int h, PParams P)
//
// Smoothes the image according to the value in P->smoothimage.
```

```
// The mask of the smoothing filter will have size 2*(P->smoothimage)+1.
//
// Parameters: float **im    - image to be smoothed
//             int w, int h  - width and height of the image
//             PParams P     - command-line parameters
// Returns:    pointer to the smoothed image
// ============================================================================

float **smoothimage (float **im, int w, int h, PParams P)
{
  float **output, **smoothfilter;
  int i, j, s;
  float sum;

  s = P->smoothimage;
  if (s)
  {
    printf ("Smoothing the image.\n");

    smoothfilter = float_array (2*s+1, 2*s+1);

    // define the smoothing filter
    sum = 0;
    for (i = 0; i < 2*s+1; i++)
      for (j = 0; j < 2*s+1; j++)
      {
        smoothfilter[i][j] = 1/(2*M_PI*s*s) *
          exp ( -0.5*((i-s)*(i-s)+(j-s)*(j-s))/(s*s) );
        sum += smoothfilter[i][j];
      }

    // normalise so that sum of the values is 1
    for (i = 0; i < 2*s+1; i++)
      for (j = 0; j < 2*s+1; j++)
        smoothfilter[i][j] /= sum;


    // convoluting the image with the filter
    output = convolution (im, w, h, smoothfilter, 2*s+1, 2*s+1);

    free_float_array (im, h);
    free_float_array(smoothfilter, 3);

    // save the smoothed file, if requested
    if (P->saveall) writefloatpgm (output, w, h, "_smoothpicture.pgm", 0);
  }
  else
  {
    output = im;
  }

  return output;
}

// ============================================================================
// float sobelfilter (float **image, int w, int h,
//                    float ***imageh, float ***imagev, float ***imagetot,
//                    PParams Params)
//
// Convolutes the image with the horizontal and vertical Sobel filter,
// and merges the two resulting edge maps together, using L2 (Eucledian) norm.
//
// Parameters: float **image     - image to be convoluted with the filters
//             int w, int h      - width and height of the image
//             float ***imageh   - pointer to horizontal edge map
//             float ***imagev   - pointer to vertical edge map
//             float ***imagetot - pointer to merged edge map
//             PParams Params - command-line parameters
// Returns:    maximum value in the merged edge map (useful for normalising)
// ============================================================================

float sobelfilter (float **image, int w, int h,
```

```
                  float ***imageh, float ***imagev, float ***imagetot,
                  PParams Params)
{
  float **sfh, **sfv;        // Sobel filters

  float maxval;
  int i,j;

  // ------------------------------------------------------------------------
  // define Sobel filters

  sfh = float_array(3,3);
  sfv = float_array(3,3);

  sfv[0][0] = sfv[2][0] = -1;
  sfv[1][0] = -2;
  sfv[0][2] = sfv[2][2] = 1;
  sfv[1][2] = 2;

  sfh[0][0] = sfh[0][2] = 1;
  sfh[0][1] = 2;
  sfh[2][0] = sfh[2][2] = -1;
  sfh[2][1] = -2;

  // ------------------------------------------------------------------------
  // apply Sobel filters

  printf ("Applying horizontal Sobel filter.\n");
  (*imageh) = convolution (image, w, h, sfh, 3, 3);
  printf ("Applying vertical Sobel filter.\n");
  (*imagev) = convolution (image, w, h, sfv, 3, 3);

  // save the filter files
  if (Params->saveall)
  {
    writefloatpgm ((*imageh), w, h, "_sobelhoriz.pgm", 0);
    writefloatpgm ((*imagev), w, h, "_sobelvert.pgm", 0);
  }

  // ------------------------------------------------------------------------
  // merging the two sobel filters
  // (ignore border areas which are distorted from using the smoothing filter)

  printf ("Merging the Sobel filters.\n");
  maxval = 0.0;
  (*imagetot) = float_array (w, h);

  for (i = Params->smoothimage + 1; i < h - Params->smoothimage - 1; i++)
  {
    for (j = Params->smoothimage + 1; j < w - Params->smoothimage - 1; j++)
    {
      (*imagetot)[i][j] =  pow((*imageh)[i][j]*(*imageh)[i][j] +
                               (*imagev)[i][j]*(*imagev)[i][j], 0.5);
      if ((*imagetot)[i][j] > maxval) maxval = (*imagetot)[i][j];
    }
  }

  if (Params->saveall) writefloatpgm ((*imagetot), w, h, "_sobelmerged.pgm", 0);

  free_float_array(sfh, 3);
  free_float_array(sfv, 3);

  return maxval;
}

// ============================================================================
// void edgesusing2ndder (float **imagefloat, float **sobel,
//                        int w, int h, PParams P)
//
// Uses the "second derivative" filter to make the edge detector more robust.
// The function convolutes the image float **imagefloat with the "Mexican hat"
// filter. The Mexican hat filter is the second derivative of the Gaussian
```

```
// filter, and convoluting the image with this filter means looking up
// the second derivative of a smoothed image. For more mathematical
// background see report. The second derivative map is then examined
// with a simple zero-crossing filter. All edges in the edgemap float **sobel
// which do not have support in a zero crossing of the second derivative,
// are then removed.
//
// Parameters: float **imagefloat - image to be processed
//             float **sobel      - the edge map - result of the Sobel filter
//             int w, int h       - width and height of the image
//             PParams P          - command-line parameters
//  Returns:   nothing
// ========================================================================

void edgesusing2ndder (float **imagefloat, float **sobel,
                       int w, int h, PParams P)
{
  float **secondder;  // second derivative map
  float **f;          // "mexican hat" filter
  int i, j, removeedge;


  // define the filter
  f = float_array(5,5);
  for (i = 0; i < 5; i++) for (j = 0; j < 5; j++) f[i][j] = mexhatsmall[i][j];

  secondder = convolution (imagefloat, w, h, f, 5, 5);

  // now look for the zero crossings of the 2nd derivative map
  // if a zero crossing does not occur, remove possible edge from the edge map
  for (i = 0; i < h; i++)
    for (j = 0; j < w; j++)
    {
      if (( i > 1) && (j > 1) && (i < h - 2) && (j < w - 2))
      {
        if (ABS(secondder[i][j] + secondder[i+1][j] + secondder[i][j+1]) ==
            ABS(secondder[i][j]) + ABS(secondder[i+1][j]) + ABS(secondder[i][j+1]))
          sobel[i][j] = 0;
      }
      else
      {
        sobel[i][j] = 0;
      }
    }
  if (P->saveall) writefloatpgm (sobel, w, h, "_edgemapimproved.pgm", 0);
  free_float_array (f, 5);
  free_float_array (secondder, h);

  return;
}

// ========================================================================
// float **thresholdandfinddirectionmap (float **sobel, int w, int h,
//                  float **sobelh, float **sobelv, float maxval, PParams P)
//
// Thresholds the edge map according to value in P->edgethreshold.
// Calculates the direction map based on the edge maps from
// vertical and horizontal Sobel filter.
//
// Parameters: float **sobel     - edge map
//             int w, int h      - width and height of the edge map
//             float **sobelh    - horinzontal edge map
//             float **sobelv    - vertical edge map
//             float maxval      - maximum value in the edge map
//             PParams P         - command-line parameters
// Returns:    pointer to the direction map
// ========================================================================

float **thresholdandfinddirectionmap (float **sobel, int w, int h,
                 float **sobelh, float **sobelv, float maxval, PParams P)
{
  float **dirmap;
```

```
  unsigned char **edgemap;  // thresholded edge map
  unsigned char **dirmapuchar;   // direction map in unsigned char array
  int i, j;

  printf ("Calculating the edge map and direction map.\n");
  dirmap = float_array (w, h);
  edgemap = uchar_array (w, h);
  dirmapuchar = uchar_array (w, h);

  for (i = 0; i < h; i++)
  {
    for (j = 0; j < w; j++)
    {
      if (sobel[i][j]/maxval*255 < P->contthreshold)
      {
        // the pixel does not correspond to an edge - set brightness to 0
        edgemap[i][j] = 0;
        dirmap[i][j] = 0;
        dirmapuchar[i][j] = 0;
        sobel[i][j] = 0;
      }
      else
      {
        // the pixel corresponds to an edge - calculate the direction
        edgemap[i][j] = 255;
        dirmap[i][j] = atan2(sobelh[i][j], sobelv[i][j]);
        if (dirmap[i][j] > M_PI_2) dirmap[i][j] -= M_PI;
        if (dirmap[i][j] < -M_PI_2) dirmap[i][j] += M_PI;
        dirmapuchar[i][j] =(unsigned char)((dirmap[i][j] / M_PI + 0.5) * 255) ;
      }
    }
  }

  if (P->saveall)
  {
    // save intermediate files, if requested
    writepgm (edgemap, w, h, "_edgemap.pgm");
    writepgm (dirmapuchar, w, h, "_edgedirections.pgm");
  }

  free_uchar_array (edgemap, h);
  free_uchar_array (dirmapuchar, h);

  return dirmap;
}

// ============================================================================
// float **accumulateinab (float **edge, int w, int h, float **dir, PParams P)
//
// Accumulates the potential centres of the circles in the (a,b)-space.
// Takes the edge map from float **edge and adds a line to the (a,b)-space
// perpendicular to the direction in float **dir. The line is drawm
// on both sides of the edge, but only in the distance between
// P->minradius and P->maxradius.
//
// Parameters: float **edge  - pointer to the edge map
//             int w, int h  - width and height of the edge map
//             float **dir   - pointer to the direction map
//             PParams P     - command-line parameters
// Returns:    pointer to the (a,b)-space
// ============================================================================

float **accumulateinab (float **edge, int w, int h, float **dir, PParams P)
{
  float **abspace;
  int i, j;
  float x, y, dx, dy;
  int x1, y1, x2, y2;  // coordinates in the (a,b)-space

  printf ("Accumulating in (a,b)-space.\n");
  abspace = float_array (w, h);
```

```
  for (i = 0; i < h; i++)
  {
    for (j = 0; j < w; j++)
    {
      if (edge[i][j] > 0)  // edge found
      {
        // calculate the starting point of the perpendicular line
        x = P->minradius * cos (dir[i][j]);
        y = P->minradius * sin (dir[i][j]);

        // calculate the step increments for drawing the line
        if ((dir[i][j] > - M_PI_4) && (dir[i][j] < M_PI_4))
        {
          dx = SIGN (x);
          dy = dx * tan(dir[i][j]);
        }
        else
        {
          dy = SIGN (y);
          dx = dy / tan(dir[i][j]);
        }

        // draw line into the (a,b)-space
        while (sqrt (x*x + y*y) < P->maxradius)
        {
          // coordinates of the potential point in the (a,b)-space
          x1 = (int) (j + x); y1 = (int) (i - y);
          x2 = (int) (j - x); y2 = (int) (i + y);
          // add a point only if the coordinates are in the image
          if ((x1 < w) && (x1 >= 0) && (y1 < h) && (y1 >= 0))
            abspace[y1][x1] += edge[i][j] / sqrt (x*x + y*y);
          if ((x2 < w) && (x2 >= 0) && (y2 < h) && (y2 >= 0))
            abspace[y2][x2] += edge[i][j] / sqrt (x*x + y*y);

          x = x + dx;
          y = y + dy;
        }
      }

    }
  }
  if (P->saveall) writefloatpgm (abspace, w, h, "_abspace1-accumulated.pgm", 0);

  return abspace;

}

// ============================================================================
// float **enhanceabspace (float **abspace, int w, int h, Params P)
//
// Concentrates the 'hot spots' by convoluting the (a,b)-space with
// a 'Mexican hat' filter. More details on the mathematical background
// regarding this filter see report.
// Further, the function thresholds the resulting (a,b)-space
// with the threshold set by -t2 parameter from command-line.
//
// Parameters: float **abspace - pointer to the (a,b)-space
//             int w, int h    - width and height of the (a,b)-space
//             PParams P       - command-line parameters
// Returns:    pointer to the enhanced (a,b)-space
// ============================================================================

float **enhanceabspace (float **abspace, int w, int h, PParams P)
{
  float **output;  // enhanced (a,b)-space
  float **e;       // enhancing filter
  float maxval;
  int i, j;

  // ----------------------------------------------------------------------
  // define the enhancing filter mask
```

```
  e = float_array(17,17);
  for (i = 0; i < 17; i++) for (j = 0; j < 17; j++) e[i][j] = mexhatlarge[i][j];


  printf ("Concentrating the hot spots in (a,b)-space. This will take a while...\n");

  // now do the convolution with the defined filter

  output = convolution (abspace, w, h, e, 17, 17);
  free_float_array (abspace, h);

  if (P->saveall) writefloatpgm (output, w, h, "_abspace2-concentrated.pgm", 0);

  // ------------------------------------------------------------------------
  // now threshold the (a,b)-space
  printf ("Thresholding the (a,b)-space.\n");

  maxval = -1;

  for (i = 0; i < h; i++)
    for (j = 0; j < w; j++)
    {
      if ((i < 9) || (j < 9) || (i > h - 9) || ( j > w - 9)) output[i][j] = 0;
          // ignoring the borders of the image
                              // where side effects from the Sobel filter occur

      if (output[i][j] < 0) output[i][j] = 0;
      if (output[i][j] > maxval) maxval = output[i][j];
    }

  if (maxval > 0)
    for (i = 0; i < h; i++)
      for (j = 0; j < w; j++)
        if (output[i][j]/maxval*255 < P->abthreshold) output[i][j] = 0;

  if (P->saveall)
    writefloatpgm (output, w, h, "_abspace3-thresholded.pgm", 0);

  free_float_array (e, 17);

  return output;
}

// ========================================================================
// void accumulateinrspace (float **abspace, float **edgemap, int w, int h,
//          unsigned char **image, PParams P)
//
// Accumulates at found centres of the circles into (r)-space.
// Calculates the sums of pixel values on circles with radii from
// P->minradius to P->maxradius, and divides the sum by the radius,
// so that circles with large radii are not preferred. Only pixels which
// direction points towards the centre are considered.
// A circle with the radius corresponding to the highest value from
// calculated sums is then drawn into the original image. If the command-line
// parameter -t3 is set to less than 255, then also circles are drawn which
// accumulation peak in (r)-space is less than maximum (255) but still higher
// than the threshold set in -t3.
//
// Parameters: float **abspace - pointer to the (a,b)-space
//             float **edgemap - pointer to the edgemap, which is used for
//                               calculating the sums for given radii
//             float **dirmap  - direction map of the edges
//             int w, int h    - width and height of the (a,b)-space
//             unsigned char **image - pointer to the original image
//             PParams P       - command-line parameters
// Returns:    nothing
// ========================================================================

void accumulateinrspace (float **abspace, float **edgemap, float **dirmap,
         int w, int h, unsigned char **image, PParams P)
{
  float *rspace;
```

```
int i, j, k, l, maxpos;
float a, b, r, step, val, spread;     // extrapolation variables
float maxa, maxb, maxr, maxv; // maximum values for extrapolation
float p1, p2, p3, p4;         // pixel values for extrapolation
float x, y, maxval;
float angle;
unsigned char **contribute;


printf ("Accumulating in r-space.\n");
rspace = (float *) calloc (P->maxradius - P->minradius + 1, sizeof (float));
contribute = uchar_array (w, h);

for (i = 0; i < h; i++)
  for (j = 0; j < w; j++)
    // if a point found in the (a,b)-space, then we found a circle centre
    // and we can accumulate in r-space for this coordinate
    if (abspace[i][j] > 0)
    {
      maxval = 0;
      maxpos = 0;
      // start accumulating for radii from P->minradius to P->maxradius
      for (k = P->minradius; k <= P->maxradius; k++)
      {
        rspace [k-P->minradius] = 0;
        // add up the points around the circle
        for (l = 0; l < k * 2.0 * M_PI; l++)
        {
          x = (float)k * cos ((float)l / (float)k);
          y = -(float)k * sin ((float)l / (float)k);

          // test whether the coordinates lie inside the image
          if (((int)(i + y + 0.5) >= 0) && ((int)(i + y + 0.5) < h) &&
              ((int)(j + x + 0.5) >= 0) && ((int)(j + x + 0.5) < w))
          {
            if (edgemap[(int)(i + y + 0.5)][(int)(j + x + 0.5)] > 0)
            {
              // if the pixel corresponds to an edge, test whether the direction
              // of the edge corresponds to a possible circle with
              // centre at (i,j)
              angle = atan2 (-y, x);
              if (angle < -M_PI_2) angle += M_PI;
              if (angle > M_PI_2) angle -= M_PI;

              // M_PI / 8 is the tolerated error for the direction
              if (ABS(angle - dirmap[(int)(i + y + 0.5)][(int)(j + x + 0.5)]) < M_PI / 8)
              {
                rspace[k - P->minradius] +=
                  edgemap[(int)(i + y + 0.5)][(int)(j + x + 0.5)];
                contribute[(int)(i + y + 0.5)][(int)(j + x + 0.5)] = 255;
              }
            }
          }

        }
        // divide the calculated sum by the radius so that large circles
        // (with larger circumference) are not preferred
        rspace[k-P->minradius] /= (float)k;
        if (rspace[k-P->minradius] > maxval)
        {
          maxval = rspace[k-P->minradius];
          maxpos = k;
        }
      }

      // threshold the r-space
      for (k = P->minradius; k <= P->maxradius; k++)
      {
        rspace[k - P->minradius] = rspace[k - P->minradius] / maxval * 255;
        if (rspace[k - P->minradius] + 0.001 < P->rthreshold)
          rspace[k - P->minradius] = 0;
        if (j - k >= 0) abspace[i][j-k] = rspace[k - P->minradius] * 1000;
```

```
      }

      // run through the r-space again to find peaks
      k = P->minradius;
      maxval = 0; maxpos = 0;
      while (k <= P->maxradius)
      {
        if (maxval < rspace [k - P->minradius])
        {
          maxval = rspace [k - P->minradius];
          maxpos = k;
        }
        if ((maxval > 0) && (rspace [k - P->minradius] == 0))
        {
          // now, we have the centre and the radius. But we want to
          // refine the coordinates and the radius first.
          if (P->refinecoordinates)
          {
            // set the parameters for the refinement
            maxv = 0; step = 0.5; spread = 3.0;
            a = i - spread;

            // run through the 3D space (a,b,r)
            while (a <= (float)i + spread)     // a coordinate
            {
              printf (" A %f\n", a);
              b = j - spread;
              while (b <= (float)j + spread)      // b coordinate
              {
                r = maxpos - spread;
                while (r <= (float)maxpos + spread)     // r coordinate
                {
                  val = 0;
                  // add up points alongside the circle
                  for (l = 0; l < 1000; l++)
                  {
                    x = (float)r * cos ((float)l / 1000.0 * 2 * M_PI);
                    y = -(float)r * sin ((float)l / 1000.0 * 2 * M_PI);
                    if ((y + a >= 0) && (y + a < h - 1) &&
                        (x + b >= 0) && (x + b < w - 1))
                    {
                      // set the value of the pixel that surround the (non-integer)
                      // coordinates that are supposed to be the centre of the
                      // circle
                      p1 = 0; p2 = 0; p3 = 0; p4 = 0;
                      angle = atan2 (-y, x);
                      if (angle < -M_PI_2) angle += M_PI;
                      if (angle > M_PI_2) angle -= M_PI;

                      if (ABS(angle - dirmap[(int)floor(y+a)][(int)floor(x+b)]) < M_PI / 8)
                        p1 = edgemap[(int)floor(y+a)][(int)floor(x+b)];
                      if (ABS(angle - dirmap[(int)floor(y+a)+1][(int)floor(x+b)]) < M_PI / 8)
                        p2 = edgemap[(int)floor(y+a)+1][(int)floor(x+b)];
                      if (ABS(angle - dirmap[(int)floor(y+a)][(int)floor(x+b)+1]) < M_PI / 8)
                        p3 = edgemap[(int)floor(y+a)][(int)floor(x+b)+1];
                      if (ABS(angle - dirmap[(int)floor(y+a)+1][(int)floor(x+b)+1]) < M_PI / 8)
                        p4 = edgemap[(int)floor(y+a)+1][(int)floor(x+b)+1];

                      // calculate the interpolation (see report for details)
                      val += ((1 - fmod(y+a, 1)) * p1 + fmod(y+a, 1) * p2) *
                              (1 - fmod(x+b, 1) +
                              (1 - fmod(y+a, 1)) * p3 + fmod(y+a, 1) * p4) *
                              fmod(x+b, 1);
                    }
                  }
//                  printf ("  a %f b %f r %f val %f\n", a, b, r, val);
                  if (val > maxv)
                  {
                    maxa = a; maxb = b; maxr = r; maxv = val;
                  }
                  r += step;
                }
```

```
                b += step;
              }
              a += step;
            }
          }  // if (P->refinecoordinates)
          else
          {
            maxr = maxpos;
            maxa = i; maxb = j;
          }
          // print out the final coordinates of the circle
          printf ("Max pos %d %d, r %d, refined to %f %f r %f\n",
                  i, j, maxpos, maxa, maxb, maxr);
          // draw the circle with radius maxval
          for (l = 0; l < maxr * 2.0 * M_PI; l++)
          {
            x = (float)maxr * cos ((float)l / (float)maxr);
            y = -(float)maxr * sin ((float)l / (float)maxr);
            if (((int)floor(maxa + y + 0.5) >= 0) && ((int)floor(maxa + y + 0.5) < h) &&
                ((int)floor(maxb + x + 0.5) >= 0) && ((int)floor(maxb + x + 0.5) < w))
            {
              image[(int)(maxa + y + 0.5)][(int)(maxb + x + 0.5)] = P->circlecolour;
            }

          }
          maxval = 0; maxpos = 0;
        }

        k++;
      }

    }

  // save the image cointaining the edge pixels that were contributing
  // to the accumulation in r-space
  if (P->saveall) writepgm (contribute, w, h, "_contributingpoints.pgm");
  free_uchar_array(contribute, h);

  return;
}

// ============================================================================
// int main(int argc, char* argv[])
//
// Parameters : command-line parameters
// Returns:     0 on success, 1 on failure
// ============================================================================

int main(int argc, char* argv[])
{

  PParams Params;

  unsigned char **image;             // image to be examined
  float **imagefloat;                // image in float values
  float **sobelh, **sobelv, **sobel; // edge maps after applying sobel filters
  float **directionmap;              // contains the direction of the edges
  float **abspace;                   // (a,b)-space

  int w, h;                          // width and height of the image
  int i, j, k, l;                    // just counters

  float maxval;

  Params = ProcessParameters (argc, argv);

  // read the image from the pgm file and convert it to float
  image = readpgm (Params->inputfilename, &w, &h);

  imagefloat = float_array (w, h);

  for (i = 0; i < h; i++)
```

```
  for (j = 0; j < w; j++) imagefloat[i][j] = image[i][j];

// ------------------------------------------------------------------------
// smooth image

imagefloat = smoothimage (imagefloat, w, h, Params);

// ------------------------------------------------------------------------
// call the sobel filters to find the edges

maxval = sobelfilter (imagefloat, w, h, &sobelh, &sobelv, &sobel, Params);

// ------------------------------------------------------------------------
// threshold the edge map, and find the direction map

directionmap = thresholdandfinddirectionmap (sobel, w, h, sobelh, sobelv,
                                             maxval, Params);

// ------------------------------------------------------------------------
// use the second derivative as an improvement of edge look-up

if (Params->use2ndderivative)
  edgesusing2ndder (imagefloat, sobel, w, h, Params);

// ------------------------------------------------------------------------
// now accumulate in (a, b)

abspace = accumulateinab (sobel, w, h, directionmap, Params);

// ------------------------------------------------------------------------
// enhance the resulting (a,b)-space, concentrate the 'hot spots'

abspace = enhanceabspace (abspace, w, h, Params);

// ------------------------------------------------------------------------
// find the centres of the spots

printf ("Finding highest values in hot spots.\n");

// remove individual dots (consider as noise)
for (i = 0; i < h; i++)
  for (j = 0; j < w; j++)
  {
    maxval = 0;
    for (k = -3; k < 4; k++)
      for (l = -3; l < 4; l++)
        if ((i + k >= 0) && (j + l >= 0) && (i + k < h) && (j + l < w) &&
            ((k != 0) || (l != 0)))
          if (abspace[i + k][j + l] > 0)  maxval++;
    if (maxval < 3) abspace[i][j] = 0;
  }

// now, call recursive function removing all pixels in the neighbourhood
// of a given pixel with lower brightness than the given pixel
// (in more detail, see the description of the function)
for (i = 0; i < h; i++)
  for (j = 0; j < w; j++)
    if (abspace[i][j] > 0) removesmallervalues (abspace, i, j, h, w);

if (Params->saveall)
  writefloatpgm (abspace, w, h, "_abspace4-loci.pgm", 0);

// ------------------------------------------------------------------------
// now I have to agrregate for all the centres in the (r)-space

accumulateinrspace (abspace, sobel, directionmap, w, h, image, Params);

writefloatpgm (abspace, w, h, "_abspace5-loci2.pgm", 0);
writepgm (image, w, h, Params->outputfilename);
printf ("\n");

// clean-up memory
```

```
  free_uchar_array(image, h);
  free_float_array (imagefloat, h);
  free_float_array (directionmap, h);
  free_float_array (abspace, h);

  free_float_array (sobelh, h);
  free_float_array (sobelv, h);
  free_float_array (sobel, h);

  free (Params->inputfilename);
  free (Params->outputfilename);
  free (Params);

  return 0;

}
```