Contour detection using a multi-scale approach and surround inhibition

Author: Reinco Hof

Rijksuniversiteit Groningen
Department of Mathematics and Computing Science
Mentor: prof. dr. sc. techn. N. Petkov
Second supervisor:prof.dr. J.B.T.M. Roerdink

Table of Contents

1. Abstract

In this thesis an algorithm for contour detection in image processing is described. The goal is to get to a contour detector that mimics the human perception of contour edges. For this purpose a performance measure is introduced, which compares the result of a contour detector with a hand drawn extended groundtruth image.
In the thesis a contour detector is gradually developed. Topics dealt with are convolution, noise reduction, gradient computation, edge thinning using non-maximal suppression and binarization. These techniques are used in a well-known edge detector, the Canny edge detector. These topics are followed by a chapter about multi-scaling and a method to suppress the strength of edges originating from texture known as surround inhibition. The performance using a single and multi-scale contour detection approach, both with and without surround inhibition, is evaluated using the performance measure. Multi-scaling proves to give no absolute performance gain, but decreases the performance spread and often the median increases. The performance gain of surround inhibition depends much on the signal-to-noise ratio. If texture and object contours are present at the same scale and the amount of texture contours is high, there can be a considerable increase.
If one looks at the output images with highest performance created by the contour detection methods described here, one notices that the object(s) present are recognizable as the object(s) shown in the input images. Unfortunately, if the input image is for example a picture of an animal or human, parts of the outline and important facial characteristics (like eyes and ears) are often missing in these output images. A human would complete the contour information using his or her experience. The contour detection algorithms described here lack this ability.

2. Introduction

The edges in an image are transitions in the intensity function of a digital image. They are located at the positions where the intensity function changes abruptly, so the locations where there is an abrupt change in gray level or color. For example if one looks at a photograph of a face, then the border of the face, the border of the eyes, nose and mouth are all edges, but also the borders of each hair.

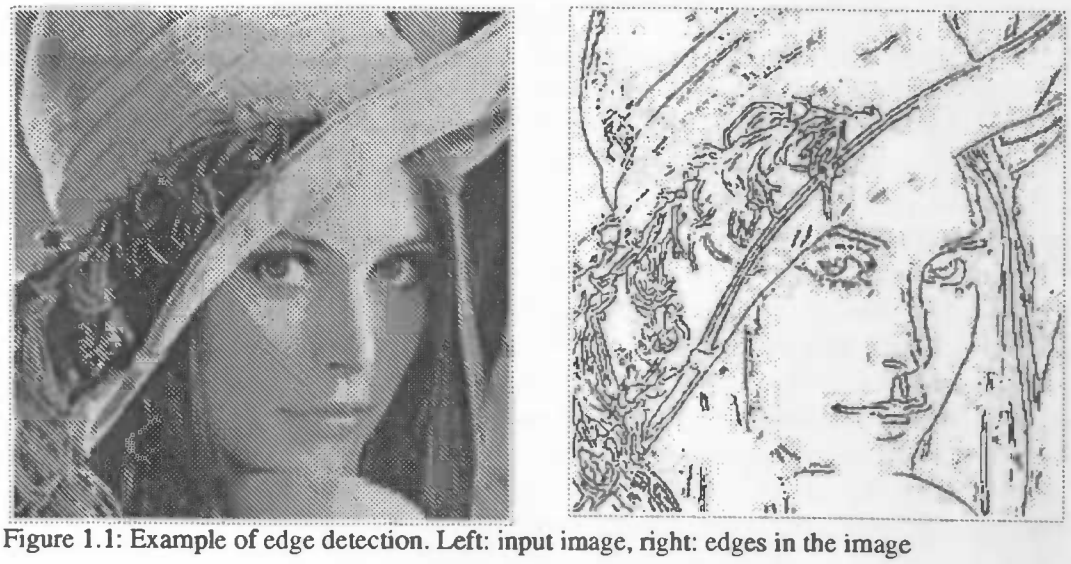


Figure 1.1: Example of edge detection. Left: input image, right: edges in the image

Edge **detection** is an important preprocessing step in image processing, as the contours of the objects of interest present in the image all generate edges. So when one locates the edges in the image one can use this information to locate and recognize the objects in the image.

Edge **detection** comprises the following three steps:

1. The first step is to filter out the noise from the image, while retaining the edges. This step is called **noise reduction**. It is needed because at noise pixels a large change in intensity occurs. This means that such pixels are regarded as edge pixels by an edge detector and the output image would contain a lot of noise.
2. Secondly the edges should be enhanced. The goal of this step is to get an image in which the edges are clearly visible, while parts of the image not containing an edge are suppressed. This is the **edge enhancement** step.
3. The last step is localization. In this step it is decided which edges in the edge enhanced image are produced by noise and which ones are really present. In the second step the edges are only roughly identified and they are multiple pixels wide. However there should only be response at the pixels where the edges are really located, so an edge **thinning** operation should be applied. As the input images used are discrete, an edge is always located on the boundary between two pixels, so one of the two pixels should be chosen as the edge's location.
   The output of this step should be a binary image, so binarization should be applied. The result is a so called **edgemap**, with zeros at the locations in the original image without an edge and ones at the edge locations. The locations with ones are called the **edge pixels**. In this image the edges are one pixel wide as a result of the thinning operation.

These are the steps to be taken for a gray level image. This means an image for which only the intensity per pixel is given. In the case of a color image, the edge detection method of choice can be applied to each color component (red, green and blue) separately, delivering three binary images, which form, when they are or-ed bitwise together, one binary image, containing ones where the edges are present and zeros at the other locations. So without loss of generality I assume that the input images of the edge detectors are gray level images.

The presence of edge pixels originating from hair as in Figure 1.1 is often undesired, so only a subset of the edges present in an image are of interest. An algorithm which finds only these edges is called a **contour detector**. In other words, a contour detector is a method that locates the edges originating from the image's contours while suppressing the edges originating from its texture (for example hair, fur or grass).
Note that a contour detector can be made by first creating an edge detector, followed by a post processing step in which the relevant edges are selected.

In the next chapter it is explained what a good contour detector is.
In the chapter hereafter convolution is explained and its role in edge detection.
Chapter 5 explains a noise reduction filter, the Gaussian filter.
Chapter 6 discusses a popular edge detection filter, the derivative of Gaussian or dG filter. The Canny edge detector, which is based on the dG filter, is also explained in that chapter.
A problem with the Canny edge detector is the scale at which to look for edges. This

problem is solved by using a multi-scale approach as explained in chapter 7.
In chapter 8 a method, called surround inhibition, that transforms an edge detector into a contour detector is discussed. The method is then used to turn the multi-scale Canny edge detector into a contour detector. In this chapter the performance using a single and multi-scale approach both with and without surround inhibition are discussed as well. At last, in the chapter 9, I will draw some conclusions.
All algorithms described in this thesis are implemented in Matlab. The source code dump of the Matlab source can be found in appendix A.

3. What makes a good contour detector

It is important to realize the difference between edge detection and contour detection. Edge detection is meant to find all the pixels in the image with an abrupt change in intensity or color, the so called edges, while a contour detector only delivers edges originating from object contours, not texture.
A good edge detector is not automatically a good contour detector as the optimal edge detector would detect all the edges present in the image not originating from noise, whereas the optimal contour detector only returns object contours. First let us define what a good edge detector is.

Often three criteria are used to check how good an edge detector is[3].

1. **Signal-to-noise ratio** or **detection criterion**. A good edge detector should have a large signal-to-noise ratio, meaning that the probability of detecting an edge present in the image is large and the probability of false positives is small. A **false positive** is an edge pixel that is present in the edgemap delivered by the edge detector, but an edge is not present in the original image at that location.
2. **Localization (of response)**. This means to exactly pinpoint the edges present in the image. Good localization is difficult, because of noise present in the image or computational errors, resulting in false localization. This means that an edge is detected, while there is no edge at all or there is a response at the wrong position close to a real edge.
3. The **resolution** of an edge detector determines the ability to distinguish two edges close to each other. A high resolution is important. This can be achieved by looking at a *small* neighbourhood of a pixel in order to determine whether it is an edge. A side effect is that the result becomes more sensitive to noise, so a good edge detector should keep this effect to a minimum.

Now these three criteria can be used to develop a good edge detector, which than can be used as the base of a contour detector. Still it is important that the results of different contour detectors can be compared objectively.
What is needed is a so called **ground truth**. A ground truth is a human created binary image containing the edges that he thinks need to be detected by a good contour detector. Of course every human would draw a slightly different binary image, but what is important is that using this ground truth one can tell how well a contour detector detects the contours from an image. By making use of the same groundtruth, the effectiveness of different contour detectors can be compared. It also becomes possible to create a contour detector that mimics the human perception of contour edges, because one has the groundtruth as an objective measure for this.

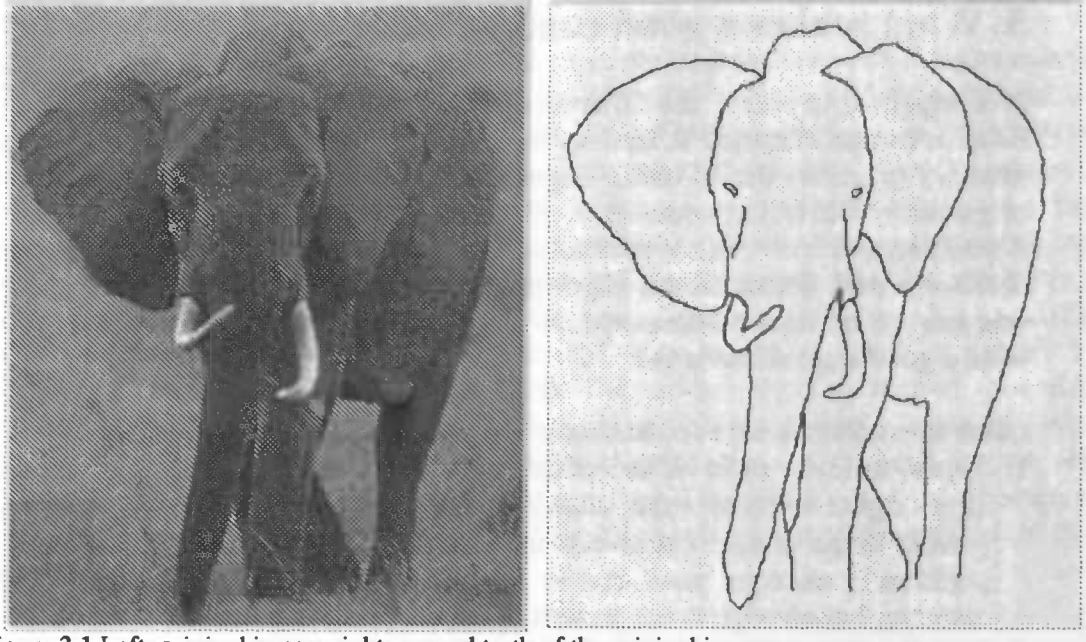


Figure 3.1 Left: original image, right: ground truth of the original image.

One method to create a ground truth in a paint program is to first scale the gray values of an image from range [0, 255] to range [0, 254]. It is assumed that 0 is the lowest and 255 the highest intensity value in the original image. The value 255, white, can then be used to draw the edges manually in the image with a one pixel wide brush. It is wise to magnify the images so that it is easier to draw them. Afterward the result can be thresholded and inverted to get a ground truth image similar to the one in the example in figure 3.1. **Thresholding** an image means that intensities below a certain intensity value get one color and all others another one.
When drawing a ground truth image, no attention should be payed to the digital imaging definition of contour detection. One is interested in mimicking the way humans recognize contours. That is the ultimate goal.

A problem is that not all edges in the image are important and even desired in the result delivered by a contour detector. For example if an animal is standing in the grass, one only wants the contours of the animal, not the edges of the grass or the animals fur, as these are not important in object recognition. So a good contour detector has the ability to detect edges originating from the objects of interest while suppressing the edges that are part of the texture of the object or its neighbourhood.
If a contour detector detects too many of the edges present in the image, it becomes hard to see where the important edges, the object contours, are.

It is hard to visually test the performance of contour detectors by comparing the groundtruth of an input image with the result of the detector on that input image. It is also important that the comparison is objective.
In [4] a performance measure for contour detectors using the groundtruth is explained: Let *EGT* be the set of contour pixels (this means the pixels marked by hand as a contour) and *BGT* be the other pixels, the so called **background pixels** of the groundtruth, and let *ED* and *BD* be the set of edge pixels and background pixels of the edgemap delivered by the contour detector respectively. The set of correctly detected contour pixels, *E*, is the set of contour pixels that is both present in the groundtruth

and the edgemap delivered by the contour detector.

$$E = ED \cap EGT \tag{3.2}$$

The set of **false negatives**, *FN*, the contours missed by the contour detector, is the set of contour pixels marked as contour pixels in the groundtruth, but not in the result of the contour detector:

$$FN = BD \cap EGT \tag{3.3}$$

Finally, the set of **false positives**, *FP*, is the set of pixels marked as contour pixels in the result of the contour detector, but which are not marked as contour pixels in the groundtruth image:

$$FP = ED \cap BGT \tag{3.4}$$

The performance, *P*, of a contour detector on a given input image with given groundtruth is the number of correctly detected contour pixels divided by the sum of this number and the number of false negatives and false positives. The amount of elements in a set, *S*, is denoted by *card*(*S*), the cardinality of *S*.

$$P = \frac{card(E)}{card(E) + card(FN) + card(FP)} \tag{3.5}$$

P always takes a values between zero and one. If all contours of the input images are detected correctly and no false positives or false negatives are present, then the performance measure returns one. If none of the contour pixels is detected correctly then the result is zero. A higher the value of *P* means that the performance of the contour detector is better on that input image with given ground truth.
Besides the performance of an edge detector, also *fn*, the fraction of correctly detected edge pixels (ie *FN/GT*) and *fp*, the amount of false positives divided by the amount of correctly detected edge pixels (ie *FP/E*) are usually calculated. It tells something about the reason why the performance is high or low. If for example the performance is low and one sees that *fn* is low, but *fp* high, then one knows that most edges are detected correctly, but that there are a lot of false positives.

There are two problems with the calculation of *E*, *FN* and *FP*. First an edge is always located on the boundary of two neighbouring pixels in a digital image, which can not be expressed in an edgemap, so the pixel left or right to the real contour is set by a contour detector. In the groundtruth the other pixel might be chosen, making the performance of the edge detector appear worse than it is.
Secondly an error is present in the groundtruth as it is a hand drawn picture. The drawer can not be expected to draw the contours very accurately.
The solution to both problems is to consider a contour pixel *p* in the groundtruth to be detected correctly if an edge pixel is present in a small neighbourhood of *p* in the edgemap. What is called *small* depends on the scale at which an edge is found: if an 'abrupt' change in intensity occurs over for example 5 pixels, then the width of the edge sloop is 5. This width is called the **edge width**. A human or edge detector may

mark any of the 5 pixels as an edge or even pixels a bit further away.
The error per pixel needs to be provided as extra information with the groundtruth. This so called **extended groundtruth** is an intensity image in which the intensity of an edge pixel equals the edge width at that pixel. The non edge pixels have intensity zero. Such an image can be constructed from the unthresholded groundtruth image *UGT* quite easily. Open the *UGT* image, which is an intensity image, in a paint program and translate it to a 24 bit color model. Also open the corresponding input image *I*. Now select a *replace color brush* and use as brush size the error margin allowed. By moving a brush of different sizes over the edges in image *I* one can find a suitable size for each pixel. As color to replace select white, which is the color of the edge pixels in *UGT*. If at a pixel a brush of w pixels wide is to be used, then use as new color *Blue=0, Green=0, Red=w*. Now move this brush over all edge pixels in *UGT* with this error margin w. The same procedure should be used for the other pixels. As last step set the color of none edge pixels to black. In the paint program I used this is not possible, so I just saved the created image and wrote a program to do this.

4. Convolution

4.1 Definition

Convolution is a strong tool in image processing. With the aid of convolution, local image transformations can be performed easily, for example noise reduction or edge enhancement. Convolution is at the base of most edge detection techniques.
As parameters this operation takes an input image and a **convolution kernel**. The input image is a matrix with at each position (or pixel) a color or intensity value. The convolution filter is also a matrix, usually smaller than the image. The convolution operator is denoted by '*'.
What in fact happens when convolution is applied, is that for each pixel (x, y) the kernel is put on top of the image such that the its center lays over pixel (x, y) in the image. The input image is now mirrored in (x, y). Each value of the convolution kernel is multiplied by the value from this mirrored image beneath it. All these results are then added, resulting in a new value of pixel (x,y) as is illustrated in figure 4.1. The kernel should be normalized, otherwise the pixels of the image can get a value that is outside the permitted range.

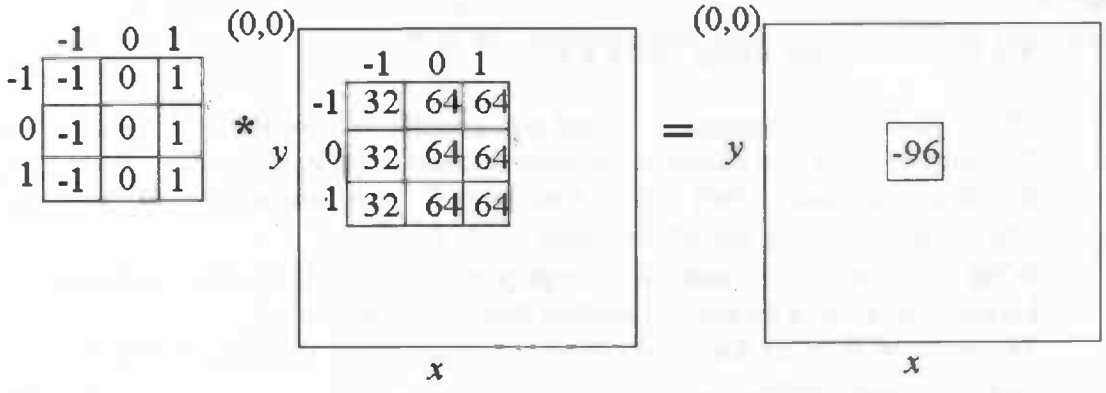


Figure 4.1: Example of convolution at pixel (x,y). Left: convolution kernel, middle: convolution kernel put over the image at pixel (x,y) with the intensity values of the pixels beneath shown in the 3x3 matrix, Right: result of applying the convolution kernel at this pixel.

The function beneath defines the convolution, described in words above. The function takes the coordinates of a pixel (x, y) as its arguments. I is the input image and K the convolution kernel. w and h are the width and height of K.

$$(I * K)(x, y) = \sum_{x'=-w/2}^{w/2} \sum_{y'=-h/2}^{h/2} I(x-x', y-y')K(x', y') \qquad 4.1$$

In case of a color image the function is applied to each color component (red, green and blue) separately, so the red, green and blue components of all the pixels in the image are regarded as three different input images. With gray level images the function is applied to the intensity values.
The convolution function 4.1 assumes that the filter's origin is its center.

A problem is what to do at the image's borders, as there pixels outside the image are involved in the calculation of the new value of those pixels. There are several solutions:

- The pixels outside the image borders are all assumed to be black. This results in a dark unnatural border at the edges, but it is the easiest solution to the problem.
- Only use the convolution function at pixels that do not involve pixels outside the image's border in their calculation. The rest of the pixels are thrown away, resulting in a image which is smaller than the input image. This arises the problem that when multiple filters are applied after another a lot of information is lost and in the worst case nothing of the image remains.
- Do not modify the border pixels or leave pixels outside the border out of the calculations. This results in strange effects near the borders. but it can sometimes be acceptable. At least the image size is not affected and there is no information loss caused by clipping.
- Repeat the image in every direction. This seems a strange thing to do, but in fact the Fourier transform, which can also be used to perform convolution as is described later on, does the same thing.
- Reflect (mirror) the image at the borders. This gives the best results, as this solution keeps the image smooth at the image borders, resulting in less artifacts near these borders. This is the solution used in this paper.

4.2 Convolution using the FFT

The problem with formula 4.1 is, that it is a rather slow method to convolve an image. For large kernels, the computation may become unacceptably slow. If the image and kernel size are both $O(n^2)$ of size then convolving a single pixel in the image takes $O(n^2)$ and convolving the whole image $O(n^4)$.

What is is needed is a method able to perform convolution fast, independent of the kernel size (at least for kernels smaller than the input image).

The solution is to do the convolution in the frequency domain, in which convolution equals multiplication.

The kernel and image are first transformed to the frequency domain using the Fast Fourier Transform (or FFT in short). Then the image and kernel matrices are multiplied element wise and at last the result is transformed back to the image domain by the reverse Fourier Transform.

As the two dimensional Fast Fourier Transform is $O(n^2 \log_2 n)$ and the pairwise multiplication $O(n^2)$, the overall time complexity becomes $O(n^2 \log_2 n)$,so the speedup achieved by doing convolution in the frequency domain is considerable.

The formula 4.3 expresses this convolution method formally. I and K are respectively the input image to transform and the convolution kernel.

$$I * K = FFT^{-1}(FFT(I)\ \ FFT(K)) \qquad 4.3$$

Note that the element wise multiplication of the Fourier transformed matrices involve multiplications of complex numbers. Also note that the kernel and image should have the same size, furthermore, the FFT requires the width and height of the matrices to be powers of two. It is best to enlarge the width of both the convolution kernel and image to the nearest power of two to the sum of the width of the kernel and image.

Similarly the height of the kernel and image should be enlarged to the nearest power of two to the sum of the height of the kernel and image. This way no artifacts remain at the borders of the image after the convolution. An alternative is to use the maximum instead of the sum. This speeds up computations and reduces memory usage considerably. The bad news is that artifacts may occur at the image borders. In the generation of the edgemaps in this thesis I used the sum. In the Matlab source (see Appendix A) the maximum is used and the code for first approach is commented out.
To meet the demands, a matrix of the required dimensions is created and is filled up with zeros. The kernel is centered in this matrix. The same strategy can be used for the input image or alternatively the input image can be repeated in every direction or reflected at the borders to get an image of the desired size. As told before at the beginning of this chapter, the last solution delivers the best results in practice and is used here.

The Fourier transform of an image is periodic. This means that the image in the frequency domain infinitely repeats itself in every direction. The Fast Fourier Transform only returns one period. Because of the choice of this period, a region from half a period to the next half is returned, the quadrants in the convolution result appear swapped.

Figure 4.2 demonstrates the problem. Each square is a period. The FFT however, delivers the image part denoted by the dotted square. Of course the figure continues infinitely in every direction.

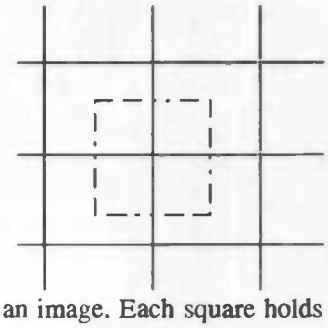


Figure 4.2: A Fourier transform of an image. Each square holds a spectral version of the image. The FFT returns the part depicted by the dotted square.

The solution is of course to swap the quadrant in the convolution result diagonally.

The steps needed to convolve an image I with a kernel K are shown in figure 4.3. The centering of the kernel in a matrix of the required size and the reflecting of the input image are shown graphically. The FFT of the two resulting matrices is calculated and the matrices are multiplied element wise. Then the inverse FFT is used to translate the result back to the image domain. In this result the quadrants need to be swapped as shown by the arrows. The black rectangle in the center with the same size as the input image contains the correct result.

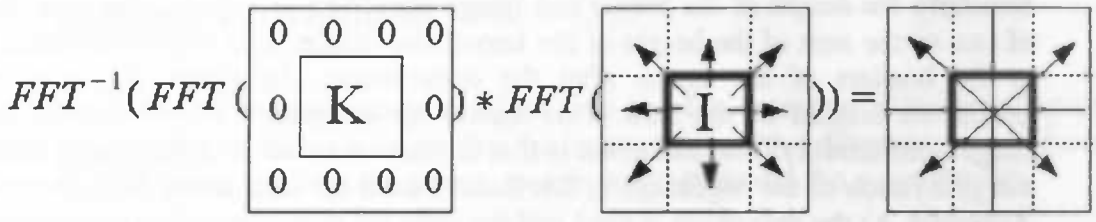


Figure 4.3: Convolution using the FFT. Left image: the kernel *K* is centered in a matrix containing only zeros. Center image: the image *I* is centered and reflected at its borders. In the result (the right most image), the four quadrants should be diagonally swapped. The center part of the result with the same dimensions as the input image *I* contains the correct result. An example is shown in figure 4.4.

As an example, let us convolve a small image with an arbitrary chosen 3x3 kernel using using the method described. All needed steps are shown in figure 4.4a to 4.4e.

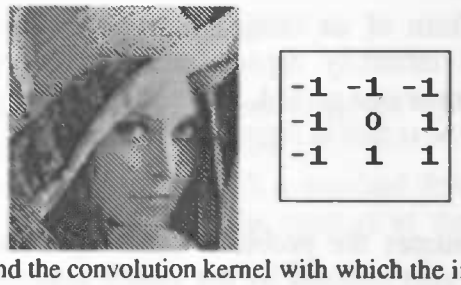


Figure 4.4a: An input image and the convolution kernel with which the image is to be convolved.

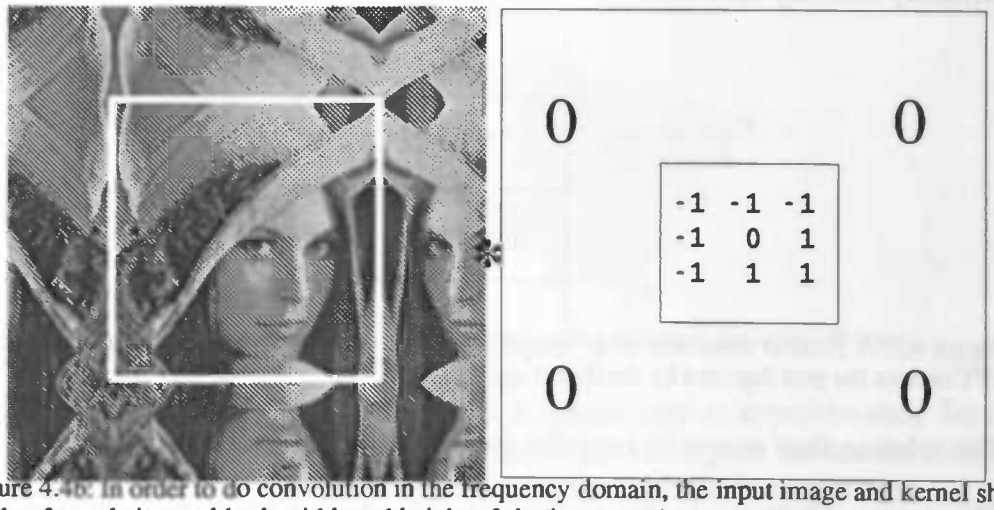


Figure 4.4b: In order to do convolution in the frequency domain, the input image and kernel should be made of equal size and both width and height of the image and convolution kernel should be powers of two. The image is enlarged by centering the image in a matrix of the required dimensions (this is shown by the white rectangle) and then the image is reflected at its borders.
For the kernel a matrix of the required size is filled with zeros and the filter kernel is centered in it.
Now the image and kernel are translated to the frequency domain, are multiplied and transformed back to the image domain.

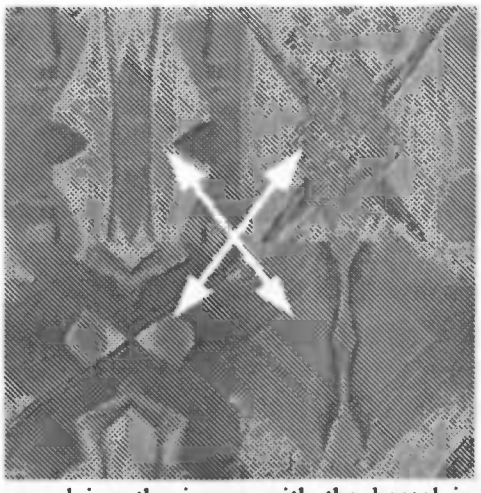


Figure 4.4c: The result of convolving the image with the kernel in the frequency domain. The quadrants appear swapped. This effect can be undone by swapping the quadrants as shown by the arrows.

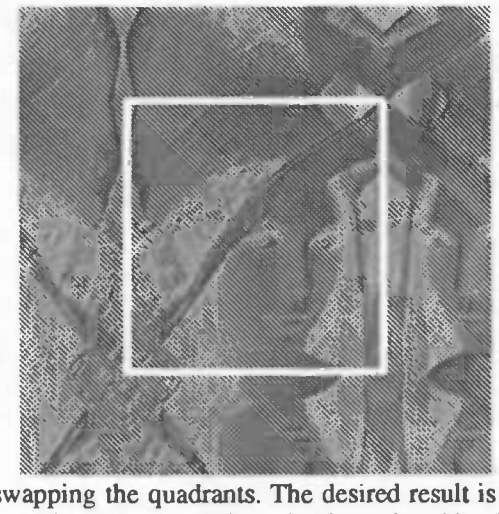


Figure 4.4d: The result of swapping the quadrants. The desired result is the center part of the image with the same dimensions as the input image. This region is enclosed by the white rectangle.

Figure 4.4e: The final result of convolving the image with the convolution kernel shown in figure 4.4a.

5. Gaussian convolution

As described in the introduction, the first step in edge detection is noise reduction. One way to achieve this is to convolve the input image with a **Gaussian** convolution kernel. This kernel smooths the image, thereby smoothing out the noise present.

A Gaussian convolution kernel template with standard deviation σ can be constructed by sampling the two dimensional Gaussian function with standard deviation σ at every integer point $(x, y) \in \mathbb{Z}$.
The two dimensional Gaussian function is defined as follows:

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \qquad 5.1$$

The function is normalized such that the area enclosed by the function and the base plane is always one, independent of the σ used.
The two dimensional Gaussian function with a standard deviation of one is displayed in figure 5.1. If a larger value is chosen, the support of the function becomes wider and it becomes less high.

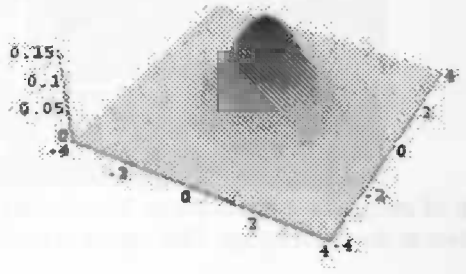


Figure 5.1: The two dimensional Gaussian function with a standard deviation of one

The two dimensional Gaussian function is almost zero at approximately 3σ from the origin. This fact can be observed in figure 5.1. In this figure the standard deviation is one and at a distance of three from the origin, the function is almost zero.
This makes it possible to construct a convolution kernel with finite size, that closely resembles the function, namely by only sampling the function at at most 3σ from the origin. The resulting convolution kernel should of course be normalized.

Gσ(−2,−2)	Gσ(−1,−2)	Gσ(0,−2)	Gσ(1,−2)	Gσ(2,−2)
Gσ(−2,−1)	Gσ(−1,−1)	Gσ(0,−1)	Gσ(1,−1)	Gσ(2,−1)
Gσ(−2,−0)	Gσ(−1,−0)	Gσ(0,−0)	Gσ(1,−0)	Gσ(2,−0)
Gσ(−2,1)	Gσ(−1,1)	Gσ(0,1)	Gσ(1,1)	Gσ(2,1)
Gσ(−2,2)	Gσ(−1,2)	Gσ(0,2)	Gσ(1,2)	Gσ(2,2)

Figure 5.2: construction of a two dimensional Gaussian convolution kernel template from a two dimensional Gaussian function where standard deviation σ equals one.

By increasing the value of σ the support of a two dimensional Gaussian function becomes wider, and the convolution kernel must be made wider as well in order to make it a good approximation of the function. So more neighbouring pixels are used to calculate the new value of a pixel p when the Gaussian convolution filter is applied and neighbouring pixels next to pixel p are weighted less. The effect is that the image is smoothed more. This way larger noise areas are removed from the image, but at the cost of blurring the image. This also means that the edges are blurred, which makes it more difficult to detect them. If the value of σ is chosen too small, the image is not blurred much, but some noise remains. It can however be shown, that among noise reduction filters, the Gaussian function offers the best solution to the problem of noise reduction versus image blurring.

Figure 5.3: A: Input image with Gaussian noise added to it. B, C, D: Input image smoothed by a Gaussian filter with a standard deviation σ of respectively 1, 3 and 5

The Gaussian convolution kernel is also popular because of its shape. Pixels further away the pixel to be smoothed are weighted less than pixels nearby in the calculation of the pixel's new value and the filter is symmetric in every direction, so that pixels at the same distance from the pixel to be smoothed all have the same influence on the result.

In figure 5.3 the Gaussian function is applied to an image with Gaussian noise added to it. In image B, a too small value of σ is used, so too little noise is removed. In image D in which the input image is smoothed by a Gaussian with a large value of σ, the noise is removed well, but the image is blurred too much. Image C offers a compromise between noise reduction and image blurring.

6. Edge detection using the derivative of Gaussian

6.1 Gradient computation

The **gradient** at a pixel in an image I equals the pair consisting of the derivative in the x-direction and the derivative in the y-direction at that pixel.
Formula 6.1 shows the calculation of the gradient of a pixel (x, y) formally. As shorthand for the derivative in the x-direction Lx is used. Similarly Ly is shorthand for the derivative in the y-direction.

$$\nabla_I(x, y) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)(x, y) = (L_x(x, y), L_y(x, y)) \qquad 6.1$$

A characteristic of the gradient of I at a pixel is that it points in the direction of the fastest change in intensity at that pixel. This fact can be used to find the edge direction, as the fastest change in intensity is always perpendicular to the edge direction. In figure 6.1 the gradient of I at a pixel together with the edge direction is displayed.

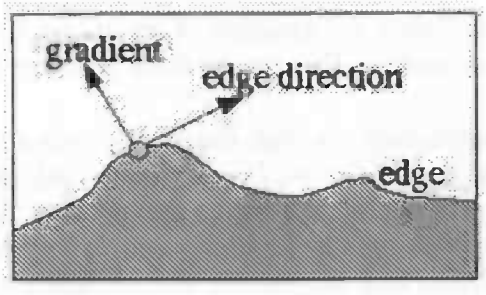
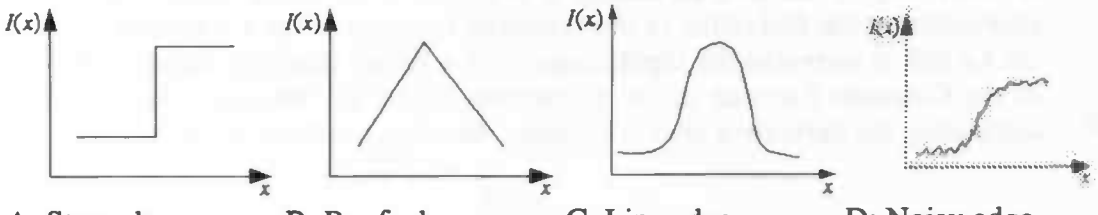


Figure 6.1: Gradient and edge direction

When looking at the edge profile in the direction of the gradient at an edge pixel, one observes a discontinuity in the intensity function. Typically one of the profiles shown in figure 6.2 can be observed.



A: Step edge B: Roof edge C: Line edge D: Noisy edge

Figure 6.2: Several edge profile kinds

If one assumes that all edges in the image are **step edges** (figure 6.2A), then the edges are located at the pixels with the locally fastest change in intensity. This means that there is a local maximum in the gradient magnitude M at those pixels.

$$M(x, y) = |\nabla(x, y)| = \sqrt{L_x(x, y)^2 + L_y(x, y)^2} \qquad 6.2$$

So if the values of matrix *M* are taken as intensity values and are displayed as an image, the locations that look bright in contrast to their neighbourhood, are the edges. Such an image is called a **(gradient) magnitude image** of the input image.
As at noise pixels also a fast change in intensity occurs, noise should first be removed from the image by a smoothing kernel. This also removes the noise from **noisy edges** (figure 6.2D), so they can be detected as well by looking at local maxima in the first derivative.
Roof edges (figure 6.2B) are typical for objects corresponding to thin lines in the image. At a roof edge there are two local maxima in the gradient magnitude: one at the rising side of the edge and one at the descending side. This means that using this method two edges will be detected, whereas there is only one present. The same holds for **line edges** (figure 6.2C).

Figure 6.3: Left: an input image with roof edges. Right: the gradient magnitude image of this image.

The conclusion is that edge detection using the gradient of the image, only works if the edges in the image are step edges, but with the right amount of smoothing also noisy edges can be detected.

As told, before calculating the gradient of an image, it needs to be smoothed in order to remove the noise present. This can be done with the Gaussian filter described in the previous chapter.
The problem remains how to find the derivatives in the *x* and *y*-direction of the Gaussian smoothed image, *Lx* and *Ly*, needed to find the gradient of the input image.
The naive approach is to shift the image one pixel in the *x*-direction and to subtract it from the image itself to get *Lx*. Similarly can *Ly* be found by shifting the image one pixel in the *y*-direction and subtracting this shifted image from the unshifted image. Because of the way *Lx* and *Ly* are obtained, the magnitude image appears shifted, so edges are detected at the wrong locations. Also there is a problem at the borders of the image as at those locations shifted data is not available and pixel values need to be invented. Finally this method of differentiation is analytically ill-posed.

A better way to calculate *Lx* and *Ly* is to convolve the input image with a kernel that approximates the derivative of the Gaussian function in the *x*-direction (figure 6.4) to get *Lx* and to convolve the input image with a kernel that approximates the derivative of the Gaussian function in the *y*-direction to get *Ly*. So smoothing the image and calculating the derivative of it in a certain direction are done in one step.

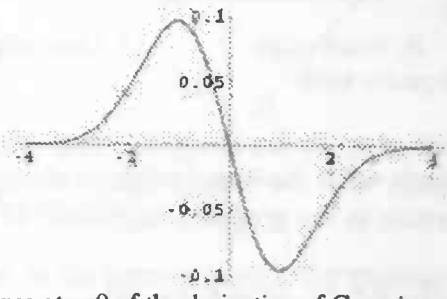


Figure 6.4: A cut along the *x*-axes at *y*=0 of the derivative of Gaussian with $\sigma = 1$ in the *x*-direction

Convolving an image I with the derivative of a Gaussian function $G\sigma$, equals taking the derivative of I smoothed with a Gaussian function. The proof of this equality is shown in equation 6.3 for the derivative in the x-direction. The idea is that in the formula $I(a,b)$ is independent of x, so the derivative of $G\sigma$ can be used instead of the derivative of the whole double integral.

$$\left(\frac{\partial I * G_\sigma}{\partial x}\right)(x,y) = \frac{\partial \iint I(a,b) G_\sigma(x-a, y-b)\, da\, db}{\partial x} = \\ = \iint I(a,b) \frac{\partial G_\sigma(x-a, y-b)}{\partial x} da\, db = \left(I * \frac{\partial G_\sigma}{\partial x}\right)(x,y) \qquad 6.3$$

Because edges are detected using the two kernels approximating the derivative of the Gaussian function in the x- and y-direction, one speaks of the **derivative of Gaussian** filter or **dG**.
This method to find Lx and Ly is shown in formula 6.4. In this formula I is the input image and $G\sigma$ is the Gaussian function with standard deviation σ.

$$L_x^\sigma(x,y) = \left(\frac{\partial G_\sigma}{\partial x} * I\right)(x,y) \\ L_y^\sigma(x,y) = \left(\frac{\partial G_\sigma}{\partial y} * I\right)(x,y) \qquad 6.4$$

The two convolution kernels are constructed the same way as the Gaussian filter (chapter five). The derivatives of the Gaussian function in both x and y-direction is nearly zero at the same distance from the origin as the Gaussian function, so to construct the kernels the derivatives are sampled at integer points at at most three times the chosen standard deviation.
For the (derivative of) Gaussian function many normalization are proposed in literature. In this paper the normalization is chosen such that when the derivative of Gaussian function is convolved with a step edge of height one, the result at the location of the edge is independent of the standard deviation used. By using the following definition the result is always 0.5 in this case:

$$\frac{\partial G_\sigma}{\partial x}(x,y) = \frac{-x}{\sqrt{2\pi\sigma^3}} e^{-\frac{x^2+y^2}{2\sigma^2}} \\ \frac{\partial G_\sigma}{\partial y}(x,y) = \frac{-y}{\sqrt{2\pi\sigma^3}} e^{-\frac{x^2+y^2}{2\sigma^2}} \qquad 6.5$$

In short the *dG* edge detector developed so far works as follows. First Lx and Ly are calculated using the dG filters. The gradient of a pixel (x, y) in the image now equal ($Lx(x, y)$, $Ly(x, y)$). The magnitudes of the gradients of the pixels in the image can be displayed. This leads to an image as can be seen in figure 6.5 in which the edges are the bright locations. So now the steps one (noise reduction) and step two (edge enhancement) in edge detection have been dealt with.

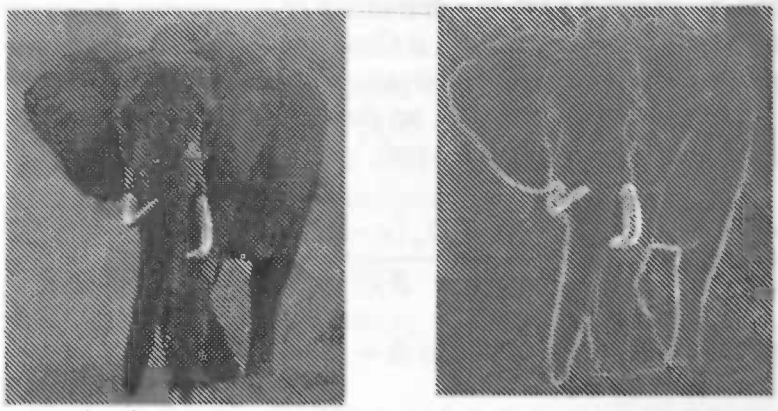


Figure 6.5: An example of edge detection with the dG. Left: input image. right: gradient magnitude image created using a dG filter with a standard deviation of one

This is not a complete edge detector yet, as the last step of edge detection (thinning and creating a binary image) still needs to be taken.

6.2 Post-processing the gradient image: the Canny edge detector

According to the introduction, the last step of an edge detector comprises of localization: thinning the edges in the image and outputting a binary image telling of each pixel whether it is an edge pixel or not.

6.2.1 Thinning the edges

Until now only the gradient magnitude information is used. In order to thinnen the edges in the gradient magnitude image *M* resulting in an gradient magnitude image *T*, in which the edges are one pixel wide, the gradient's direction at a pixel is important as well.

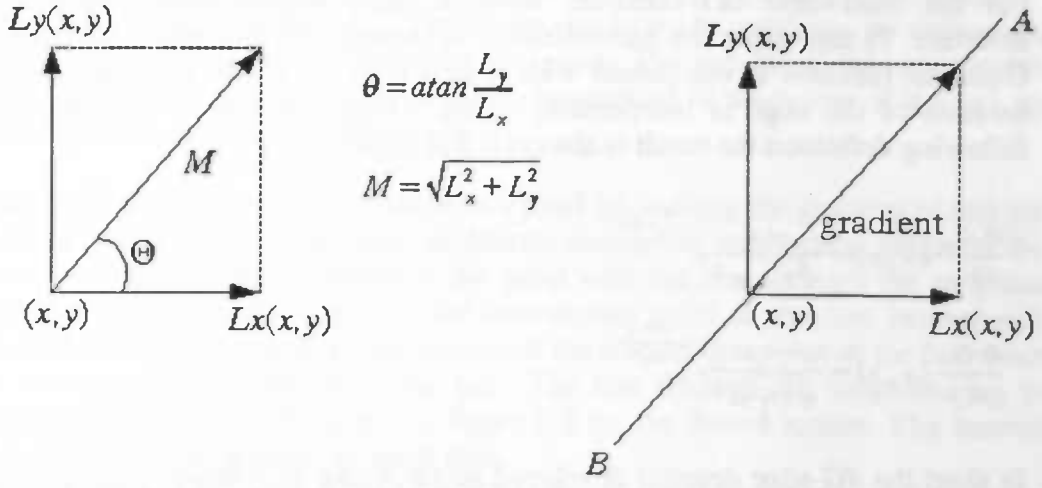


Figure 6.6: Left: the gradient at pixel (*x*, *y*) with its direction Θ and magnitude *M*. Right: look in both directions of the gradient of a pixel (*x*, *y*). If the magnitude at this pixel is larger than that of both *A* and *B* then the pixel is a candidate edge pixel.

An edge is present at a pixel in the input image if the gradient magnitude image has a local maximum at that pixel in the direction of the gradient, so what needs to be done

is to look in the direction of the gradient at some point A and in the opposite direction at a point B and if the magnitude of the pixel's gradient is larger than A and larger or equal to B, the pixel is an edge pixel. It states larger or *equal* because an edge is always located at the border between two pixels. If no noise is present, both pixels have the same gradient magnitude, so in that case the local maximum is two pixels wide. The specified condition ensures that exactly one of the two pixels is marked as an edge pixel.

If a pixel is not an edge pixel then its gradient magnitude is set to zero else it remains unchanged by the thinning step. This method for thinning the edges in the image is called **non maximal suppression**. It thins the wide ridges around local maxima to one pixel wide. Below the idea is shown by means of a formula.

$$T(x,y) = \begin{cases} M(x,y) & \text{if } M(x,y) > M(A_x(x,y), A_y(x,y)) \,\&\, M(x,y) \geq M(B_x(x,y), B_y(x,y)) \\ 0 & \text{otherwise} \end{cases} \quad 6.6$$

The question remains which points one should chose for point A and B and how to determine their magnitudes. Two solutions to this problem are described here. The first solution works best on artificial images, the second one works best on digital photographs.

6.2.1a Non-maximum suppression by comparing neighbouring pixels

In case an image is artificial, which means that the image is not a digitalized version of a real world object, one can choose the pixels next to pixel (x, y) in the direction of the gradient and in its opposite direction as respectively A and B. The magnitudes of A and B then simply are the magnitudes of these two pixels.
In this case A can be found as follows: first the angle between the gradient at pixel (x, y) and the horizontal line through the pixel's center is calculated. Now a horizontal, vertical and two diagonal lines through its center are defined, resulting in 8 half lines originating in the pixel's center. Each of these half lines passes through a different neighbouring pixel.

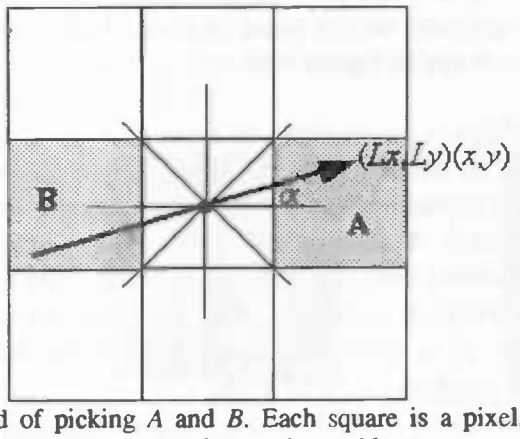


Figure 6.7: Simple method of picking A and B. Each square is a pixel. The center pixel is (x,y). $(Lx,Ly)(x,y)$ is the gradient of pixel (x,y). A and B are denoted by gray squares.

In figure 6.7 these halflines are shown by solid thin lines. These halflines all have a certain angle with the horizontal line through (x, y). If the halfline with angle closest to passes through neighbouring pixel N, then A equals N. B is the pixel opposite to A.

In other words if A equals $(x+dx, y+dy)$ then B equals $(x-dx, y-dy)$.
In formula 6.7 the selection of A and B is shown formally. Note that if $M(x,y)$ is zero, a division by zero occurs in the calculation of α. As there can not be a local maxima in the gradient magnitude image at a pixel with gradient magnitude zero, the only thing needed is to check beforehand if $M(x,y)$ is zero. If so it remains zero else the steps shown in formula 6.7 should be performed.

$$A(x,y) = (A_x(x,y), A_y(x,y)) = (x + dx(x,y), y + dy(x,y))$$
$$B(x,y) = (B_x(x,y), B_y(x,y)) = (x - dx(x,y), y - dy(x,y))$$

$$\alpha = acos(Lx(x,y)/M(x,y))$$
$$dx = 1 \text{ if } \alpha < 3/8\pi$$
$$dx = -1 \text{ if } \alpha > 5/8\pi$$
$$dx = 0 \text{ otherwise}$$
$$dy = sign(Lx(x,y)) \text{ if } 1/8\pi < \alpha < 7/8\pi$$
$$dy = 0 \text{ otherwise}$$
$$sign(v) = -1 \text{ if } v < 0 \text{ and } 1 \text{ otherwise}$$

6.7

This method of non-maximum suppression works fine with artificial images, but most of the time images used in image processing are digitalized photographs. The value of each pixel in this digitalized image is calculated by averaging the values of the original photograph in the neighbourhood of the pixel. From this one can conclude that for this kind of images, it is better to try to reconstruct the continues image. Then one can select A and B in this continuous image. The following method for selecting A and B uses this approach.

6.2.1b Non-maximum suppression using linear interpolation

An image can be seen as a finite rectangle consisting of 1x1 squares. Within each square the intensity is the same. The image can be made continues by only giving the center of each pixel the intensity of the pixel. The values of the other points are interpolated. The gradient of the pixel is also located in the center of the 1x1 square. This results in the image in figure 6.8.

One could take arbitrary points close to a pixel (x, y) along the gradient of this pixel as point A and B, but to make the calculations easier, the intersection points of the line extension of the gradient vector at the pixel with the box through the neighbouring pixels' centers is used. A then is the intersection point of the line extension in the direction of the gradient with the box and B the intersection point of the line extension in the opposite direction with the box. The box through the neighbouring pixels' centers of pixel (x, y) is shown in figure 6.8 by the dotted square. The intersection points A and B are marked by small dots.
The magnitudes at A and B are found by linear interpolation between the magnitudes of the pixels' centers on the box next to the points A and B. Note that the magnitudes of these pixels' centers are the magnitudes of these pixels themselves. This gives a fairly good approximation, which is not computationally intensive, but which clearly gives better results than just taking the arched pixel's as A and B suggested by the simple approach.

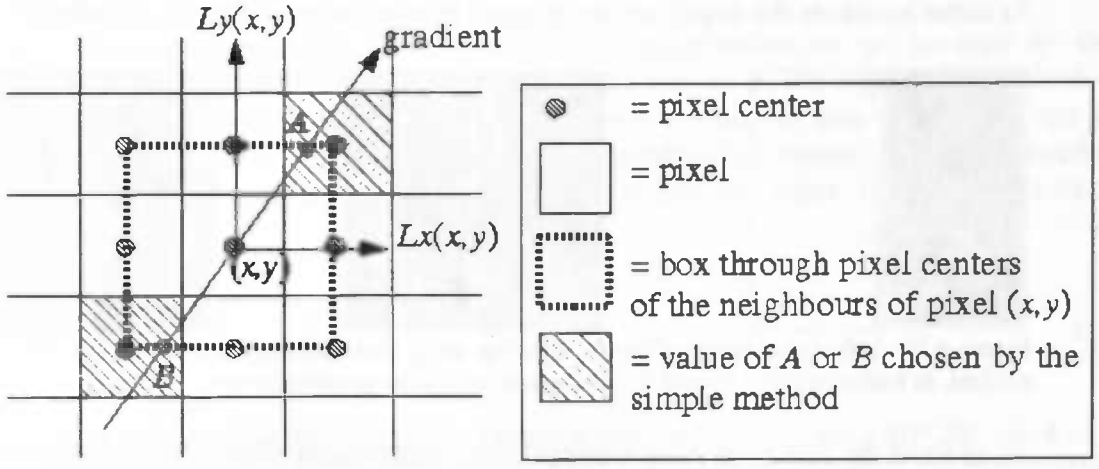


Figure 6.8: Turning a digital image into a continues one. The pixel (x, y) is displayed with its gradient. Only at the pixels' centers (the dark gray dots) the pixel take their intensity value. The values of the other points are found by means of interpolation.

Some implementation problems remain.
First as usual there is a problem at the border of the image, as those pixels do not have neighbours in all directions. For simplicity the pixels outside the image are assumed to have magnitudes of zero, probably resulting in a false positive. At the borders however, there is too little information to detect edges, so there is no real solution to the problem.
Secondly there is the special case in which $Lx(x, y)$ or $Ly(x, y)$ are both zero, in which case the dotted box in figure 6.11 is not intersected. In this case however, the gradient magnitude is zero as well. As the gradient magnitude is always at least zero, there can not be a local maximum in the gradient magnitude image at such a pixel, so the magnitude of the pixel should be set to zero in T.

Note that this second method does not work well on artificial images (for example an image consisting of polygons) as with these kind of images the assumption that a pixel's value is an average of intensities does not hold. As a result holes might appear at the edge corners or a line is not thinned enough, because a pixels gradient magnitude is compared to the wrong gradient magnitude values. These side effects be seen in respectively figure 6.9 and 6.10.

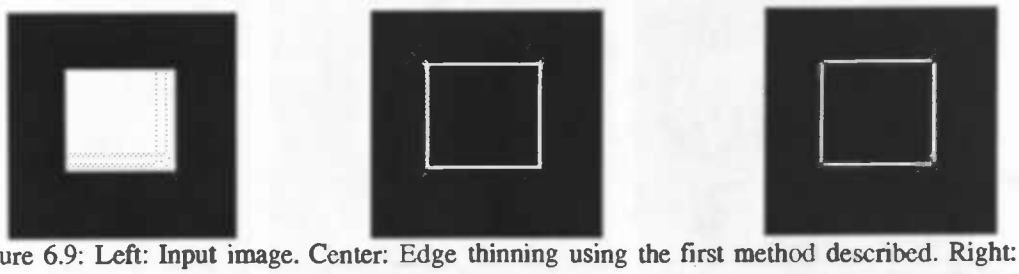


Figure 6.9: Left: Input image. Center: Edge thinning using the first method described. Right: Edge thinning using the second method.

In figure 6.9 the first and second method are applied both to a white square on a black background. The first method correctly detects the edges, but the other one leaves holes at the corners of the white square.
In figure 6.10 edge detection using the two methods is performed on an image in

which an edge at which the change in intensity occurs at an ever larger scale is done. At some locations the edges are not thinned enough when the second method is used.

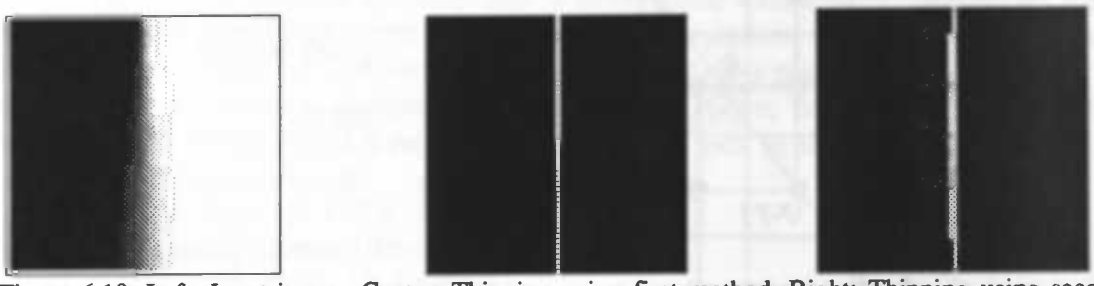


Figure 6.10: Left: Input image. Center: Thinning using first method. Right: Thinning using second method. In both cases σ=4 is used in the construction of the gradient image.

So in short the first method of non-maximal suppression should be used with artificial images and the second solution, which makes use of linear interpolation, for digital photographs.

In case the input image is a color image, three gradients images are calculated, one per color component. As the edge thinning algorithms described here work on a single gradient image only, these three images should first be combined into one. This can be done by using per pixel the gradient with maximal magnitude.
Note that applying edge thinning to each of the three gradient images separately and combining the results afterward is not wise as a single edge may give a response at slightly different locations in each of the gradient images. Now if the edge thinning results are combined, one edge may give a response at multiple pixels. This approach is also less computation efficient as the edge thinning algorithm is run three times instead of just once.

In figure 6.11 the effect of edge thinning using non-maximum suppression can be seen. By thinning the edges it is possible to exactly pinpoint their location. Also some edges appearing as one wide edge in the gradient magnitude image, for example the tusks, are actual two edges close to each other, as can be seen in the gradient magnitude image with thinned edges, so by means of edge thinning the resolution increases.

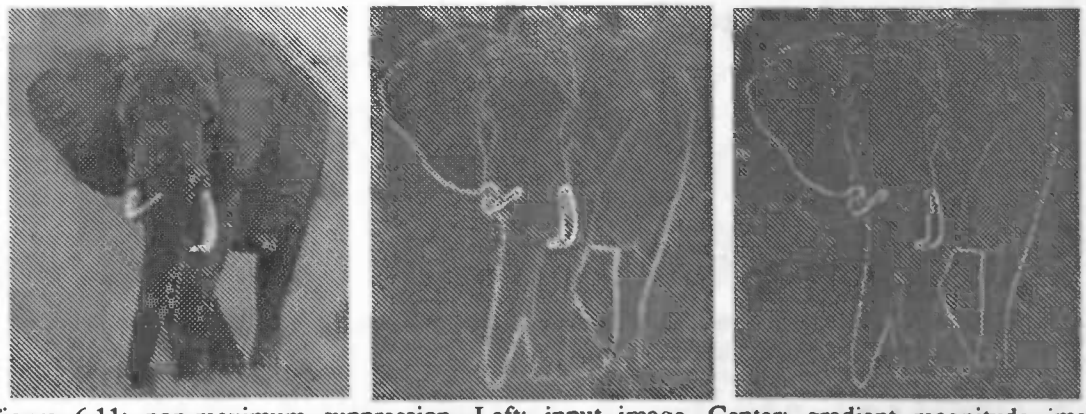


Figure 6.11: non-maximum suppression. Left: input image. Center: gradient magnitude image constructed using the derivative of Gaussian filter with σ=1. Right: non-maximum suppression applied to the center image. In the center and right image the brightness is enhanced in order to make the edges more visible.

6.2.2 Producing a binary image

Now the magnitude image, *T*, in which some magnitudes are set to zero by the thinning step, is thresholded to produce a binary image, *B*, the edgemap, in which a value of zero means that no edge is present and a value of one means an edge is present in the original image at that location. **Thresholding** means that values smaller than a certain magnitude t are set to one value and the other values are set to a different value.

$$B(x,y) = \begin{cases} 0 & \text{if } T(x,y) < t \\ 1 & \text{otherwise} \end{cases} \qquad 6.8$$

It is wise to first scale the magnitude image to the range [0, 1], such that the value of t can be chosen independently of the input image used (and the range of the magnitudes in the magnitude image).
Thresholding the image also removes false positives as those stand out in the image by the fact that their magnitudes are usual close to zero. If the threshold value t is chosen too small, not enough false positives are removed. If it is however too large all false positives are removed, but also parts of the edges that are really present, resulting in fragmented edges, so a small value, for example 0.15 should be chosen.

In figure 6.12 an image with thinned edges is thresholded with a threshold value of 1/8, the value which delivered the best results in practice. As can be seen the method does not distinguish between edges that origin from object contours and edges that origin from the texture in the image. The result is that in parts of the grass in which the elephant is standing also edges are detected. Also some edges in the elephant are not continues. This is caused by the thresholding, but if a lower threshold value is used too much unimportant edges are detected.

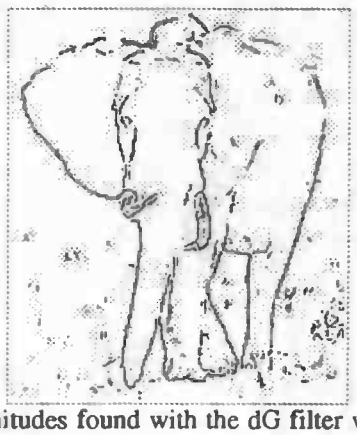


Figure 6.12: The gradient magnitudes found with the dG filter with a standard deviation of one. The edges are thinned and thresholded with a threshold of 1/8 resulting in the displayed edgemap.

In seems that producing an edgemap by thresholding the magnitude image is not a very good approach. The solution is to use **hysteresis thresholding**.
First the image is thresholded with a high threshold value, *th*. This leads to an edgemap with fragmented edges, but with little false positives produced by noise. The gaps in the contours are now filled as follows: all pixels next to a pixel already marked as an edge in the edgemap are marked as edges as well if their gradient

magnitude is at least *tl*. This is called **edge tracking**.
The reason why this works is because normally weak responses originate from noise, but if a pixel with a rather low response is next to one with a high response its more likely that the pixel is an actual edge pixel. Implementations differ in the way which neighbouring pixels are checked in the determination if a pixel should be marked as an edge pixel. Possible choices are looking only above, beneath and next to the current pixel, the so called **4-neighbourhood**, or to look in the diagonal direction as well, the **8-neighbourhood**.

```
Step 1: Thresholding with th:
for each pixel (x,y) do
  if T(x,y) < th then B(x,y)=0 else B(x,y)=1
end for

Step 2: Edge tracking using threshold tl:
repeat
  Bold = B;
  for each pixel (x,y) do
    if T(x,y) >= tl) then
      for each neighbouring pixel (nx, ny) of (x,y) do
        if B(nx, ny) == 1 then
          B(x,y) = 1;
        end if
      end for
    end if
  end for
until Bold == B;
```

Figure 6.13: Code fragment for performing hysteresis thresholding

The edge tracking step leads to a new edgemap and the step can be repeated till the edgemap does not change any more. The value of *tl* can be quite low, so that big fluctuations may occur in the contours of the image and edge fragmentation becomes low. *th* can be large, such that edges resulting from noise are not visible in the edgemap. In figure 6.13 a code fragment is presented to perform hysteresis thresholding on an image T with thresholds *tl* and *th*, resulting in a binary image B.

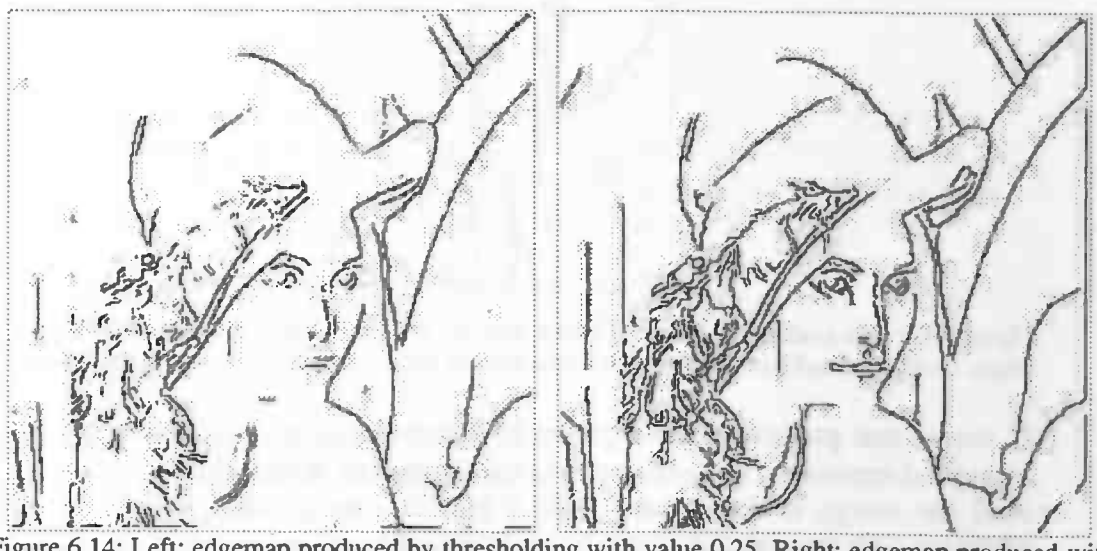


Figure 6.14: Left: edgemap produced by thresholding with value 0.25. Right: edgemap produced with hysteresis thresholding with tl =0.125 and th=0.25.

In figure 6.14 the results from thresholding with and without hysteresis are shown. It is clear that the result from hysteresis thresholding, shown in the right image, is better as the contours are less fragmented. To make a better comparison possible the threshold value with normal thresholding and the value of *th* are chosen the same.

The resulting edge detector described here, which first produces a gradient magnitude image with the aid of the derivative of Gaussian filter, followed by non maximal suppression and hysteresis thresholding is called the **Canny edge detector** after the inventor of this edge detection technique[5].
In figure 6.15 the Canny Edge Detector is applied. The right image is the final result as can be found using the method previously described.

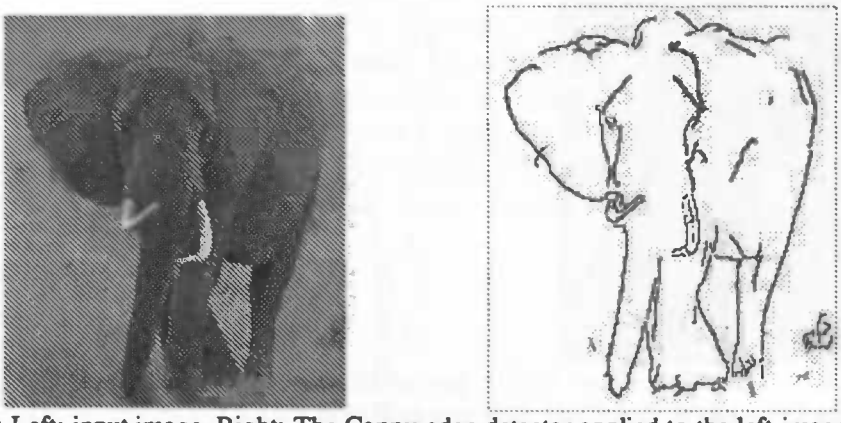


Figure 6.15: Left: input image, Right: The Canny edge detector applied to the left image (using σ=1, *tl* = 0.08, *th*=0.20).

6.3 Performance of the Canny Edge Detector

The Canny Edge Detector is optimal with respect to the three criteria mentioned in chapter three. The optimal edge detection filter Canny found using his localization criteria was very close to dG[5]. He however claimed that although it is not optimal, the dG filter could be used instead, because it is close to the optimal filter he has found and an effective implementation of the dG filter was already available. The dG filter approximates his filter by only a 20% error, so he claimed that although his edge detector was not optimal, the error introduced was acceptable.
As described in [3] Canny's measure needs modification as in the construction of his measure he uses a substitution which is not always applicable and he neglects the fact that one edge can give rise multiple responses. The result of the modification is a performance measure, which when used to find the optimal filter, results in the dG filter.
So the Canny Edge Detector is optimal with respect to the three criteria from chapter three and measure introduced in[3].

The goal however is not to create an edge detector, but a contour detector. Let us check the performance of the Canny edge detector with the performance measure from chapter 3. This way one can see how well this edge detector performs as a contour detector. In figure 6.18 from left to right an input image, its groundtruth and the Canny edge detector's best result (this is the result with the parameters σ, *th* and *tl* chosen such, that the performance *P* is maximal).

The result looks a lot like the desired result (the center image). Almost no contours originating from the texture, the grass, are present, but a lot of contours from the elephant are missing! The problem is that in order to get little false positives (edge originating from the grass), a high value of *th* is used. Although the amount of correctly detected edges is smaller, the performance will be higher with a larger value of *th*, as there are far more edges originating from texture than object contours in the image. The real problem is not the choice of parameters, but the fact that the Canny Edge Detector is meant to detect edges, not contours; it does not distinguish between edges originating from object and texture contours. A solution is presented in chapter 8.

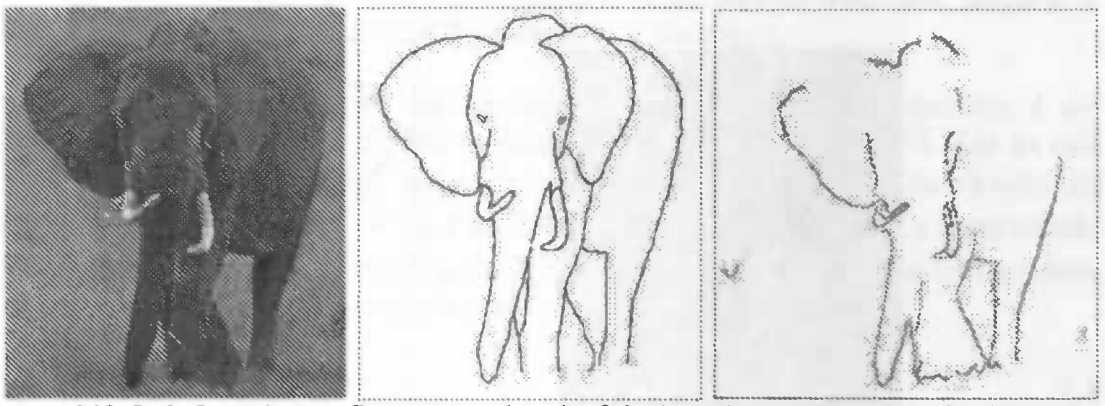


Figure 6.18: Left: Input image. Center: ground truth of the input image. Right: Result with the best performance using the Canny Edge Detector applied to the top left image: σ=2, tl = 0.13, th=0.27. The performance P (see chapter 3) is 0.46.

7. Multi-scaling

A problem with the Canny edge detector is the choice of the standard deviation, σ. If a small value is chosen, an edge is detected at a certain pixel if there is a large enough change of intensity in a small neighbourhood of that pixel. If however a large value of σ is used, pixels at which the intensity increases or decreases more gradually are identified as edges. In this case multiple edges close to another present in an image are detected at the wrong location or not at all. Usually one does not prefer one scale at which edges are present above another; one wants to detect both narrow and wide edges correctly.
In figure 7.1 the problem is illustrated. The input image is shown on the left. The center and right image are created by applying the Canny edge detector to it. The center one shows the result when a small standard deviation is used, whereas for the construction of the right image a larger value is used. In the first case the edges of the white bar are detected, but the edge with a more gradual change in intensity is not. In the latter case the wider edge is detected correctly, but the edges of the narrow white bar are not. So with a single value of σ, not all edges can be detected.

Figure 7.1: Left: Input image with one narrow white bar and one edge with a more gradual change in intensity. Center: Edgemap computed with the Canny edge detector using a small σ (σ=1). Right: Edgemap calculated using a larger σ (σ=11). In both cases *tl*=0.04 and *th*=0.20.

This is not merely a theoretical problem; in photographic images it can also occur. For example objects out of camera focus are vague, resulting in wide edges, whereas objects in focus are sharp and have narrow edges. In figure 7.2 an example is presented. The image on the left is the input image. The center and right ones are both constructed using the Canny edge detector. In the center image a small value of sigma is used. The contours from the hair of the gnu in the foreground are detected, but the feet and shadow of the gnu standing in the background are only partially detected. If a larger value of sigma is used, the opposite is true.

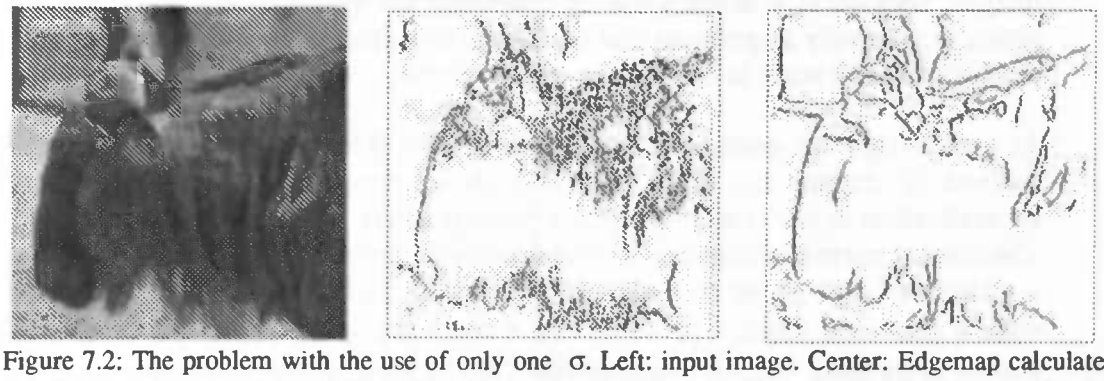


Figure 7.2: The problem with the use of only one σ. Left: input image. Center: Edgemap calculated with the Canny edge detector with σ=1. Right: Edgemap created with σ=4. In both cases *tl*=0.04 and *th*=0.20.

A solution to the problem is to combine the edgemaps created using different values for the standard deviation by means of the bit-wise OR function. At first glance this looks like a good solution, but in fact it has several disadvantages. The most important of them is that for different values of σ used, edges are detected at slightly different locations. The result is that, in the combined edgemap some edges give multiple responses.
A better solution would be to combine the gradient images created using different values of σ to get a single gradient image. This is the approach taken here. The algorithm is also more computationally and memory efficient than the first solution. Furthermore, this method delivers a gradient magnitude image, supplying us with both edge location and direction information. With the bit-wise OR solution a separate algorithm is needed to find the direction of the edges.

First multiple gradient images for an input image with intensity function I are produced using n values for σ. This is done as described in chapter 6.1. Let us call them $\nabla_0 I, \ldots, \nabla_{n-1} I$ where $\nabla_i I$ is the gradient image created with σ_i The σ's used are defined by the parameters σ_l, σ_h and k. The values used are chosen on a **logarithmic** scale in the range $[\sigma_l, \sigma_h]$. Using more values would only increase computation time without increasing the detector's performance. The σ's used are:

$$\sigma = k^0\sigma_l, \ldots, k^{n-2}\sigma_l, \sigma_h \qquad 7.1$$

where n is the largest natural number such that $k^{n-2}\sigma_l < \sigma_h$.
The gradient images can be merged into one gradient image G by using the value at a pixel (x, y) of the gradient image that has the maximum gradient magnitude at location (x, y) among the constructed gradient magnitude images:

$$G(x, y) = max\{\nabla_i I | i \in \{0, \ldots, n-1\}\} \qquad 7.2$$

The **winner** or σ used at a pixel (x, y), $W(x, y)$, is defined as:

$$W(x, y) = i \text{ if } G(x, y) = \nabla_i I(x, y) \qquad 7.3$$

Unfortunately this does not work without proper normalization of the gradient images, because if a larger value of σ is used, the gradient magnitude value of edge pixels is relatively larger than that of edge pixels detectable with a smaller value. As a result the wrong scale is used at some edge pixels.

At a step edge the gradient magnitude is always 0.5. This is the way the gradient is defined in chapter 6.1. It is clear that in order to make the smallest σ win, the normalization to use is a division by a number larger than one.
The correct normalization can be found using an input image consisting of a black and a white half. Per group of pixel rows the image is blurred by convolving the group with a Gaussian filter. The values of σ used are taken using the same concept as shown in formula 7.1. An image constructed this way is shown in figure 7.3.
Now the image is convolved with several derivative of Gaussian filters in order to find the gradient images $\nabla_0 I, \ldots, \nabla_{n-1} I$ as described before. The same values of σ

should be used as in the construction of the input image. The normalization factor per gradient image should be chosen such that when the gradient images are combined as shown in figure 7.1, at each edge pixel the σ that yields the highest gradient magnitude equals the one used to blur the pixel in the input image. This condition should hold as in this case the convolution kernel used to find an edge has the same width as the edge itself.

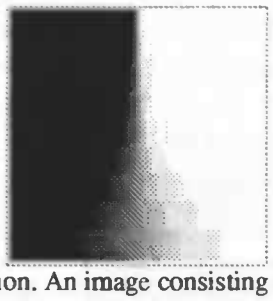


Figure 7.3: Test image for edge detection. An image consisting of a white an black half is divided into equally sized groups of pixel rows. Then each group is blurred with a Gaussian filter. The values of σ used are defined by defined formula 7.1 where $\sigma_l = 1, \sigma_h = 16, k = \sqrt{2}$.

In order to find the correct normalization factor, several normalizations of the gradient images are tried. Now one can check for which normalization the condition holds at the center column; the column containing an edge. I found that the condition holds if the following normalization is used: if a gradient image is created using standard deviation σ, it should be normalized by dividing it by $\sqrt{\sigma}$. The combined gradient image is computed as follows:

$$G(x,y) = max\{\frac{\nabla_i I}{\sqrt{\sigma_i}} | i \in \{0,\ldots,n-1\}\} \qquad 7.4$$

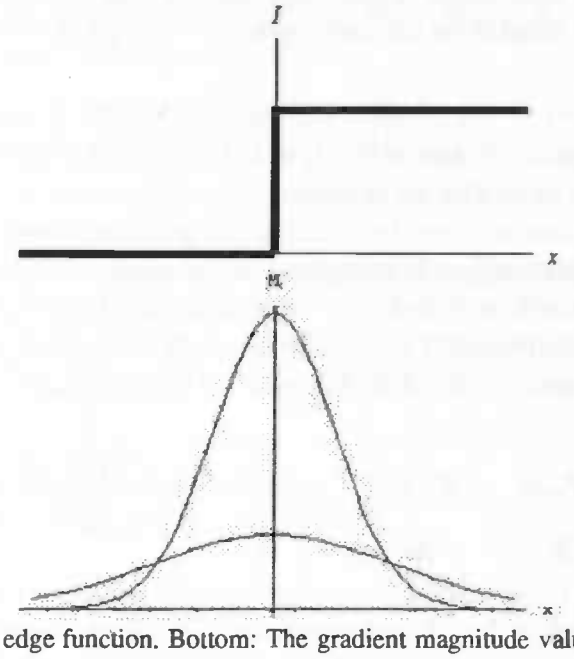


Figure 7.4: Top: A step edge function. Bottom: The gradient magnitude values for the step edge for both a low and high σ, respectively shown by the function with highest and lowest maximum.

A problem is that close to a narrow edge a small σ wins, but at some distance from the edge larger σ give a larger gradient magnitude value, because the derivative of Gaussian decreases more slowly for higher σ. This is shown in figure 7.4.

The effect is that at some distance from the edge, the gradient magnitude |*G*| is still quite large. This means that in *G*, the spread in magnitudes becomes lower. This is a bad thing, because now pixels with a smaller change in intensity get a relatively higher gradient magnitude than was the case when only the winning σ at such pixels was applied. So after thresholding *G*, a lot of edges are present that have a relatively low gradient magnitude in all gradient images used in the construction of *G* as is demonstrated for a picture of a gnu in figure 7.5; In the binarization the vague edges from the sand show up. Such edges are of course undesired in the edge detection result.

Figure 7.5: Left: Input image. Right: Edge thinning and hysteresis thresholding applied to the combined gradient image of the left image, with $\sigma_1 = 1, \sigma_n = 8, k = \sqrt{2}, tl = 0.04, th = 0.20$.

The solution is to check whether a pixel with winner σ_i has indeed a local maximum in gradient magnitude image $|\nabla_i I|$ at that pixel in direction $\nabla_i I(x, y)$. If not the gradient $G(x, y)$ should be set to zero.
Concretely this can be done by creating per gradient image $\nabla_i I$ a gradient magnitude image with thinned edges *Ti* using one of the methods described in sub-chapter 6.2.1. Now if σ_i wins at $G(x,y)$ and $T_i(x, y)$ is zero then $G(x,y)$ should be set to zero else it should remain unaffected. Let us call the result $G^*(x,y)$. As G^* is only nonzero at edge pixels, the problem is solved. In formula 7.5 this approach is shown:

$$G^*(x,y) = \begin{cases} G(x,y) & \text{if } \exists (0 \le i < n : G(x,y) = \dfrac{\nabla_i I(x,y)}{\sqrt{\sigma_i}} \wedge T_i(x,y) > 0) \\ (0,0) & \text{otherwise} \end{cases} \qquad 7.5$$

Another problem is that if two edges are present close to each other, for example if the input image is a three pixel wide white bar, the σ equaling the edge width gives a response at the correct location, but the filter kernels used with higher σ's overlap both

edges. This causes a shift of the local maxima in the gradient magnitude image. As a result it is possible that one edge gives multiple responses in the combined gradient magnitude image, $|G|$. The problem is illustrated in figure 7.6.

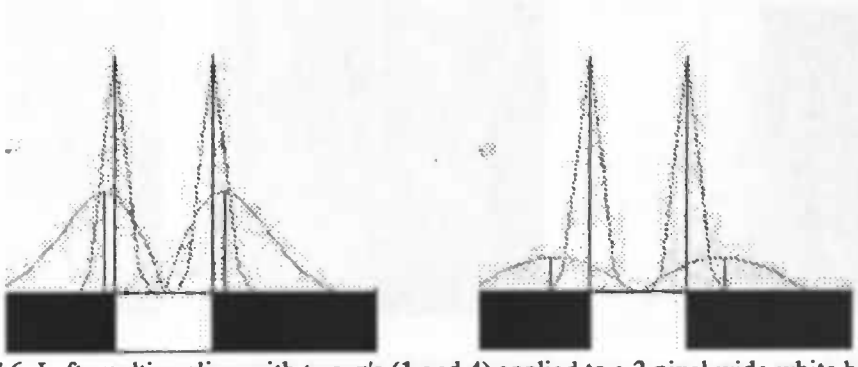


Figure 7.6: Left: multi-scaling with two σ's (1 and 4) applied to a 3 pixel wide white bar. A horizontal cut of both the normalized gradient magnitude images created using σ=1and σ=4 are shown in the same figure in gray together with the maxima in black. Right: the same, but now with σ=1 and σ=8.

Because of the normalization by $\sqrt{\sigma}$ σ=1 gives a much larger response than σ=4. At the location of the maxima produced by σ=4, the normalized gradient magnitude value found with σ=1 is larger, so in this case the shifted edges are removed, which is shown in figure 7.6a.
In general however, this is not true, for example if instead of σ=4, σ=8 is used as shown in figure 7.6b.
One way out is to use standard deviations on a logarithmic scale as shown in formula 7.1 with $k=\sqrt{2}$. Now if multi-scaling is applied to the white bar, σ=1 gives a local maximum at the edge location, which is maximal among all σ's used, so this edge kept. σ= $\sqrt{2}$ gives a shifted maximum, but at that location σ=1 still gives a larger gradient magnitude, so the shifted edge will not appear. σ=2 gives a shifted local maximum even further away, but now σ= $\sqrt{2}$ has a larger gradient magnitude there, so again the shifted edge is removed, etcetera.

One can observe that at two edges with different width, but with the same overall change in intensity, the wider edge has a smaller gradient magnitude than the other one. As the change in intensity is the same, one would want the two edges to have equal gradient magnitudes.
This goal can be achieved by another normalization. The normalization factor, for which the condition, that edges with equal overall change in intensity have the same gradient magnitude, holds, can be found using different normalizations with the test image from figure 7.3. It turns out that a multiplication of $G^*(x,y)$ by $\sqrt{(\sigma_{w(x,y)})}$ achieves this goal:

$$\bar{G}^*(x,y) = G^*(x,y)\sqrt{\sigma_{w(x,y)}} = (\nabla_{w(x,y)} I)(x,y) \qquad 7.6$$

As shown in formula 7.6, a normalization by this factor cancels out the normalization applied earlier, so one can conclude that the unnormalized winning gradient should be used instead of the normalized one.

The 'problem' and its solution are shown in figure 7.7. In the center image the gradient magnitude becomes ever smaller when the the edge becomes wider. In the right image, in which the suggested normalization is applied, this is not the case.

Figure 7.7: Left: input image. Center: |G*| without normalization, Right: $|\bar{G}^*|$.

As a final remark to multi-scaling, note that as an edge is not detected at precisely the same location at different scales, one edge might give multiple responses in |G*|, so even if edge thinning is applied to the individual gradient images, edge thinning still needs to be applied to the combined gradient image as well.

In short the multi-scaling algorithm described in this chapter consists of the following six steps:

---

1) Create n gradient images, $\{\nabla_0 I, \ldots, \nabla_{n-1} I\}$ using the following values for σ: $k^0\sigma_i, \ldots, k^{n-2}\sigma_i, \sigma_a$ for given values of σ_i and σ_a where n is the largest natural number such that $k^{n-2}\sigma_i < \sigma_a$ and k equals $\sqrt{2}$.
2) Thinnen the edges in these n gradient images. This results in n gradient magnitude images $\{T_0(x, y), \ldots, T_{n-1}\}$.
3) Compute the σ to use per pixel (x, y). This is the smallest σ, which gives a normalized gradient magnitude which is maximal among all normalized gradient magnitudes computed at (x, y):

$$W(x, y) = k::\sigma_k =$$

$$min\{\sigma_i \mid i \in \{0, \ldots, n-1\} \wedge \frac{|\nabla_i I|}{\sqrt{\sigma_i}} = max\{\frac{|\nabla_j I|}{\sqrt{\sigma_j}} \mid j \in \{0, \ldots, n-1\}\}\}$$

4) Combine the gradient images into one by using per pixel the *unnormalized* winning gradient:

$$G(x, y) = \nabla_{W(x,y)} I$$

5) If the σ_i used at pixel (x, y) in G does not detect an edge at that location (this means $Ti(x,y)$ is zero) then set the gradient at (x, y) to zero. The result is a gradient image G*:

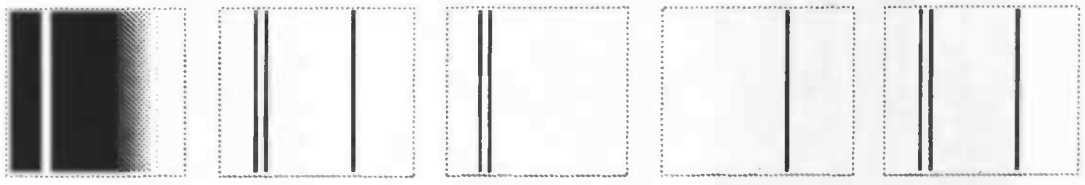
$$G^*(x,y) = \begin{cases} G(x,y) & \text{if } \exists (0 \le i < n :: G(x,y) = \nabla_i I(x,y) \wedge T_i(x,y) > 0) \\ (0,0) & \text{otherwise} \end{cases}$$

6) Apply edge thinning to G^* and threshold the result using hysteresis thresholding.

Figure 7.7: the multi-scaling algorithm

In figure 7.8 this multi-scaling algorithm is applied to the image of the narrow and

wide edge introduced earlier in this chapter. As can be seen the multi-scaling approach correctly detects both narrow and wide edges present in an input image.



2 edges | groundtruth | *P*=0.67,*fp*=0, *fn*=0.33, σ=1, *tl*=0.04 *th*=0.20 | *P*=0.33,*fp*=0, *fn*=0.67, σ=11, *tl*=0.04 *th*=0.20 | *P*=1,*fp*=0, *fn*=0, σ={1,11}, *tl*=0.04 *th*=0.20

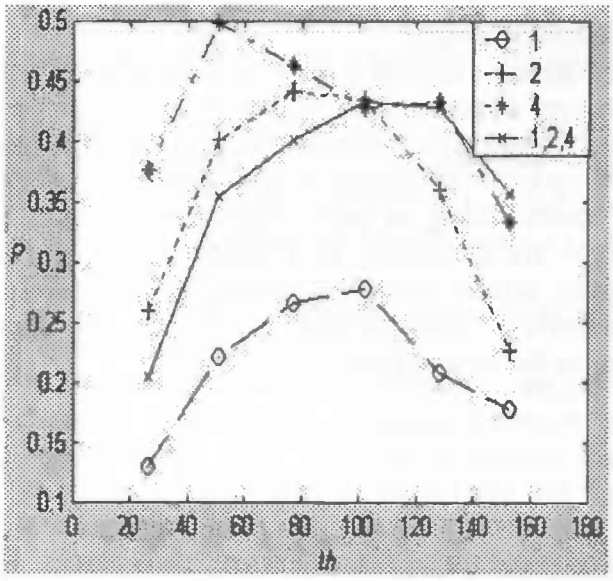
Figure 7.8: First: input image with one white bar and one edge with a more gradual change in intensity. Second: groundtruth. Third: The edgemap of the left image calculated with the Canny edge detector using a small σ. Fourth: The edgemap of the left image calculated using a larger σ. Fifth image: multi-scaling using the same combination of the σ 's.

In figure 7.9 and 7.10 the algorithm is applied to digital photographs.
Note that the performance *P* is influenced a lot by the chosen threshold *th*. This is caused by the fact that in general there are far more false positives than correctly detected edges in the edge detector's result, so hysteresis thresholding using a higher value of *th* will decrease *fp* (this is the amount of false positives divided by the amount of correctly detected edge pixels) considerably. As a result the performance will increase. Because the amount of false negatives becomes ever higher, the performance(*th*) function has a maximal though.
In general the fraction of correctly detected edge pixels, *fn*, is lower if multiple σ's are applied instead of just one if *th* is kept constant. This means that less edge pixels are missed. The number of false positives is high however, because the false positives found using the different σ's show up in the combined gradient image. As a result the performance often decreases if multi-scaling is used. The false positives usually originate from texture. Such edges are undesired, so a method is to be found to remove the edges originating from texture from the result, thereby reducing the amount of false positives considerably. One method which does exactly this is surround inhibition. It is described in the next chapter.

The input image bear_2 and its groundtruth

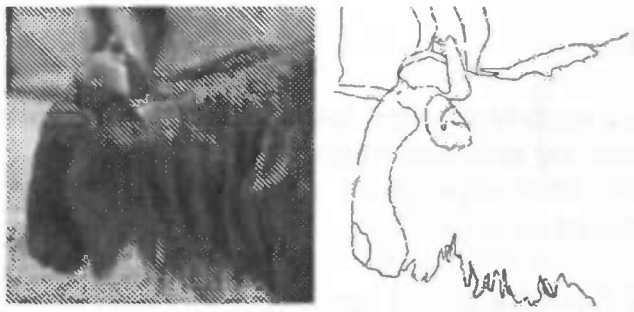
σ=1 σ=2 σ=4 σ={1,2,4}

Best results contour detection results achieved with σ=1, 2 and 4 and multi-scaling with all three σ's



Th	sigmas		fn	fp	P
0.1	1	:	0.221311	6.409023	0.129984
0.2	1	:	0.390933	2.875309	0.221378
0.3	1	:	0.525928	1.643260	0.266479
0.4	1	:	0.637504	0.832487	0.278463
0.5	1	:	0.753095	0.732385	0.209095
0.6	1	:	0.802777	0.553859	0.177801
0.1	1	:	0.134660	2.692248	0.259884
0.2	1	:	0.257946	1.140667	0.401884
0.3	1	:	0.423386	0.520814	0.443806
0.4	1	:	0.496153	0.316069	0.434632
0.5	1	:	0.589997	0.334557	0.360547
0.6	1	:	0.760120	0.241283	0.226755
0.1	1	:	0.164771	1.454436	0.377115
0.2	1	:	0.240716	0.687596	0.498846
0.3	1	:	0.422048	0.427786	0.463385
0.4	1	:	0.524590	0.228008	0.428916
0.5	1	:	0.543326	0.119414	0.433058
0.6	1	:	0.658247	0.075869	0.333116
0.1	1,2,4	:	0.129140	3.715136	0.205616
0.2	1,2,4	:	0.209602	1.548571	0.355397
0.3	1,2,4	:	0.354466	0.943509	0.401185
0.4	1,2,4	:	0.474988	0.400446	0.433863
0.5	1,2,4	:	0.519739	0.254267	0.427996
0.6	1,2,4	:	0.626129	0.135123	0.355892

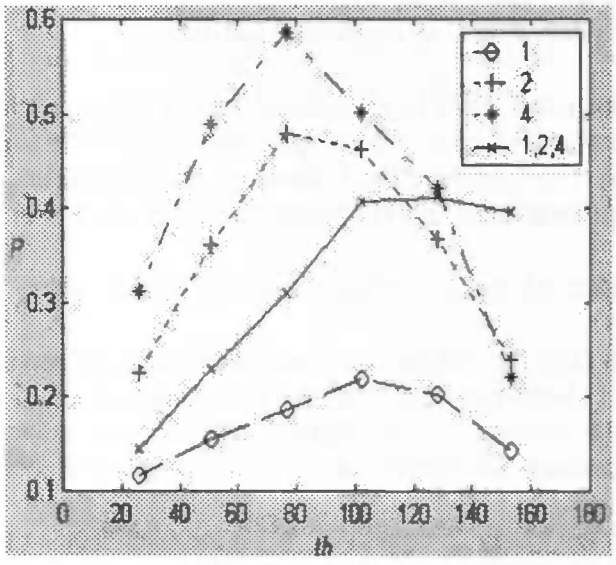
Figure 7.9: The canny edge detector is applied to the image bear_2 for σ=1, 2 and 4. Multi-scaling is applied with σ={1,2,4}. *tl* is always 0.04. For *th* six different values are used: {0.1, 0.2, 0.3, 0.4, 0.5, 0.6}. For this input image *fp*, *fn* and *P* are shown in a table for all σ's and thresholds used. Per (combination of) σ the threshold that gives the best performance is marked gray in this table. Left to the table a graph shows per (combination of) σ a plot in which the high threshold is plotted against the performance.



Input image gnu_2 and its groundtruth

σ=1 σ=2 σ=4 σ={1,2,4}

Best results contour detection results achieved with σ=1, 2 and 4 and multi-scaling with all three σ's



Th	sigmas		fn	fp	P
0.1	1	:	0.118878	7.433214	0.116712
0.2	1	:	0.288705	5.054969	0.154778
0.3	1	:	0.361572	3.778534	0.187095
0.4	1	:	0.435821	2.766538	0.220312
0.5	1	:	0.568918	2.588639	0.203733
0.6	1	:	0.735387	3.217910	0.142918
0.1	2	:	0.040679	3.383903	0.225922
0.2	2	:	0.127567	1.626528	0.360653
0.3	2	:	0.259084	0.736674	0.479305
0.4	2	:	0.464850	0.291882	0.462852
0.5	2	:	0.617496	0.112545	0.366717
0.6	2	:	0.754739	0.096618	0.239583
0.1	4	:	0.100118	2.097433	0.311654
0.2	4	:	0.197472	0.796506	0.489580
0.3	4	:	0.304107	0.269012	0.586161
0.4	4	:	0.452212	0.169430	0.501265
0.5	4	:	0.564179	0.079746	0.421183
0.6	4	:	0.773499	0.082825	0.222330
0.1	1,2,4	:	0.042457	5.867808	0.144673
0.2	1,2,4	:	0.093997	3.261988	0.229056
0.3	1,2,4	:	0.168641	2.027791	0.309536
0.4	1,2,4	:	0.274684	1.081677	0.406440
0.5	1,2,4	:	0.438981	0.646011	0.411441
0.6	1,2,4	:	0.529028	0.403774	0.395719

Figure 7.10: The canny edge detector is applied to the image gnu_2 for σ=1, 2 and 4. Multi-scaling is applied with σ={1,2,4}. *tl* is always 0.04. For *th* six different values are used: {0.1, 0.2, 0.3, 0.4, 0.5, 0.6}. For this input image *fp, fn* and *P* are shown in a table for all σ's and thresholds used. Per (combination of) σ the threshold that gives the best performance is marked gray in this table. Left to the table a graph shows per (combination of) σ a plot in which the high threshold is plotted against the performance.

8. Surround inhibition

In the introduction I told that a contour detector can be constructed from an edge detector by an extra step in which the important edges (those originating from object contours) are selected and the other edge pixels are removed from the result. **Surround inhibition**[4] is such a step.

Surround inhibition means that if an edge pixel is surrounded by others, its strength is reduced, whereas the strength of isolated edge pixels remains the same. So if the surround inhibited result is thresholded, the non-isolated edge pixels will not show up. The **strength** of an edge pixel (x,y) is a measure of the importance of an edge. Before surround inhibition is applied it equals the gradient magnitude $M(x,y)$. Afterward it will be lower or equal to it.
The idea is that edges close to another usually origin from texture, meaning that they are unimportant, whereas isolated edges originate from object contours. So with surround inhibition one can distinguish object contours from texture contours.

The method is inspired by various psychological and neuro-physiological studies which have shown that the response to a line or edge is reduced by the adding other stimuli in the neighbourhood. In the primary visual cortex orientation selective cells exist, of which 80% exhibit this mostly inhibitive effect. It is referred to as **non-classical receptive field (non-CRF) inhibition** by neuro-physiologists, but in this chapter it is referred to in a more general way as **surround inhibition**.

One can reduce the strength of an edge pixel (x,y), surrounded by others as follows. The weighted values of the surrounding edge pixels' gradient magnitude value are added. This results in a term $t(x,y)$. t should have the following characteristics:

- If more edge pixels are present near (x,y), $t(x,y)$ should be higher than if less edge pixels are present.
- If the gradient magnitude value of neighbouring pixels is higher, $t(x,y)$ should become higher as well.
- Edge pixels further away from (x,y) should have an ever decreasing influence. At a certain distance this influence becomes neglectable. This distance should be a parameter, r, to t. r should be chosen low if edges in the texture are close to another. If the edges in the texture are further apart a larger value is needed in order to suppress such edges.
- Edge pixels very close to (x,y) are likely to be part of the same edge line as that of (x,y), so such pixel should not have influence on $t(x,y)$. What is called very close depends on the scale at which edges are present. Let us call the maximal distance from (x,y) at which edge pixels have no influence on $t(x,y)$, r'. If edges in the texture are further apart, edges are present on a larger scale, and via versa, so r' is directly related to r.
  For the special case $r \approx 5 * r'$, the donut shaped region that has influence on $t(x,y)$,approximately equals the inhibitive area of the oriented selective cells in the primary visual cortex.

Figure 8.1: Influence zone of the inhibitive effect. The area enclosed by the inner ring and the area outside the outer ring have neglectable influence on $t(x,y)$.

One solution is to make use of the **difference of Gaussian** or ***DoG*** function as defined in formula 8.1. *k1* and *k2* are constants with the condition that both values should be larger than zero and *k1* should be larger than *k2*. The Gaussian function with standard deviation σ is denoted by G_σ.

$$DoG_\sigma(x,y) = G_{k_1\sigma}(x,y) - G_{k_2\sigma}(x,y) = \frac{1}{2\pi(k1\sigma)^2} e^{-\frac{x^2+y^2}{2(k1\sigma)^2}} - \frac{1}{2\pi(k2\sigma)^2} e^{-\frac{x^2+y^2}{2(k2\sigma)^2}} \qquad 8.1$$

By mapping the negative values to zero, one gets a function that is zero near the origin. This function is normalized with the L1-norm. The result is the function ***HDoG***:

$$HDoG_\sigma(x,y) = \frac{H(DoG_\sigma(x,y))}{\|H(DoG_\sigma(x,y))\|_1} \text{ where } H(x) = max(0,x) \qquad 8.2$$

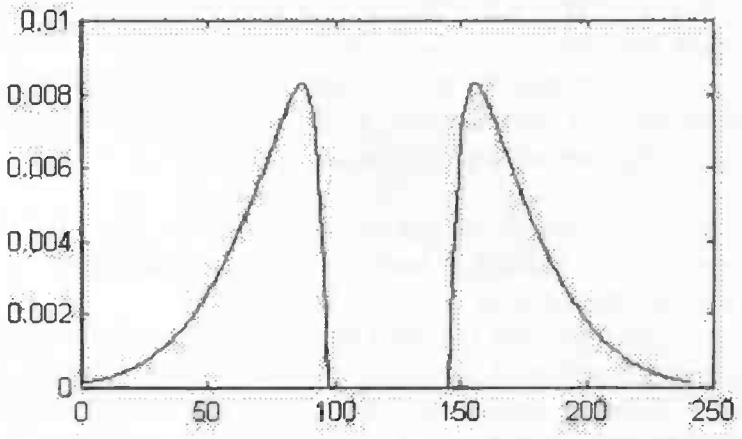


Figure 8.2: A cut of the *H*(*DoG*) function where σ=1, *k1*=4 and *k2*=1 along the *x*-axes at *y*=0.

The area that is zero near the origin depends mostly on the factor *k2* σ. By increasing it, this area becomes larger. By increasing *k1* σ, the support of the function becomes wider. This makes the convolution of this function with *M* a good candidate for *t*.
Note that *k1* σ plays the role of *r* and *k2* σ the role of *r'*. By using the *k1*=4 in the first Gaussian and *k2*=1 in formula 8.1, the same influence zone as shown in figure 8.1 is achieved. The normalization with the *L1*-norm ensures that *t* is independent of the location and size of the inhibitive area.
At each pixel (*x*, *y*) σ should equal the scale at which an edge is present, so the

standard deviation used in the computation of the gradient at that pixel.
The idea is to convolve M with multiple $HDoG\sigma$ convolution kernels. The σ's to be used are those applied in the computation of M. Suppose these are $\sigma_1, \ldots, \sigma_n$ then the results are $t_{\sigma_1}, \ldots, t_{\sigma_n}$ with

$$t_\sigma = M * HDoG_\sigma \qquad 8.3$$

If at a pixel (x, y) the σ used is $W(x, y)$ then $t(x, y)$ is defined as follows:

$$t(x, y) = t_{W(x,y)}(x, y) \qquad 8.4$$

Note that the $HDoG\sigma$ function is almost zero at approximately $\sigma(3*k1+k2)$ from the origin, so a convolution kernel for the $HdoG\sigma$ function can be constructed by sampling this function at every integer point at at most this distance from the origin.

Now the new edge strength of pixel (x,y) can be found by subtracting a fraction of $t(x,y)$ from the gradient magnitude of (x,y), $M(x,y)$. As edge strengths must be positive numbers, the negative outcomes are mapped to zero. If this scheme is applied to all pixels then we get a magnitude image C_α as defined by formula 8.5. This function performs the surround inhibition.
α controls the strength of the inhibition of the surround on the gradient magnitude. By increasing α, the reduction effect is increased.

$$C_\alpha(x, y) = H(M(x, y) - \alpha\, t(x, y)) \qquad 8.5$$

If there is no texture near a given point (x,y), $t(x, y)$ is zero and $C_\alpha(x, y)$ equals $M(x,y)$. If other edges are present, $\alpha\, t(x, y)$ is nonzero and $t(x, y)$ may even become larger than $M(x, y)$, in which case the pixel's new strength becomes zero.
As the orientation of surrounding edges is not taken into account, this kind of inhibition is called **isotropic suppression**.

The surround inhibition algorithm can be applied in any edge detector, but here the Canny edge detector, applied at multiple scales, is used, because this edge detector is optimal (see sub chapter 6.3).
Because surround inhibition involves the weighted gradient magnitude of *edge* pixels, the place at which surround inhibition should be inserted in the Canny edge detector, is after edge thinning (and before thresholding). So in short we get the following contour detection algorithm:

```
Step 1: Edge enhancement and edge thinning using multiscaling.
        This yields a gradient magnitude image M in which the
        edges are thinned.
Step 2: Apply surround inhibition. This yields a
        gradient magnitude image C as defined by formula 8.4.
Step 3: Apply hysteresis thresholding. The result is a
        binarization B.
```

Figure 8.3: The contour detector combining multi-scaling and surround inhibition.

In figure 8.4 the effect of surround inhibition using the algorithm from figure 8.3 is shown on an image of a gnu. In this figure each of the parameters α, k1 and k2 are changed in turn to demonstrate their effect.
One can observe by comparing image *B, C* and D in figure 8.4 that by increasing α, ever more edges originating from texture (the hair of the gnu) are thresholded away, because a larger fraction of *t is* subtracted from *M* in formula 8.4.
The parameters *k1* influences the maximal distance at which edges cause inhibition of an edge pixel. At the same time the influence of edge pixels nearby this pixel decreases, so with a low value edges very close to another are suppressed, while with a larger value, edges are suppresses if they are somewhat further apart. To a human observer however, this is hardly noticeable. The effect of the *k1* parameter can be seen by comparing the images *C* and *E*, which use respectively *k1*=4 and *k1*=16, while using the same values for others parameters.
With *k2* the minimum distance from an edge pixel at which other pixels have influence is affected. A larger value increases this distance. The effect of k2 can be seen by comparing image *C* and *F*. Again the effect is minimal.

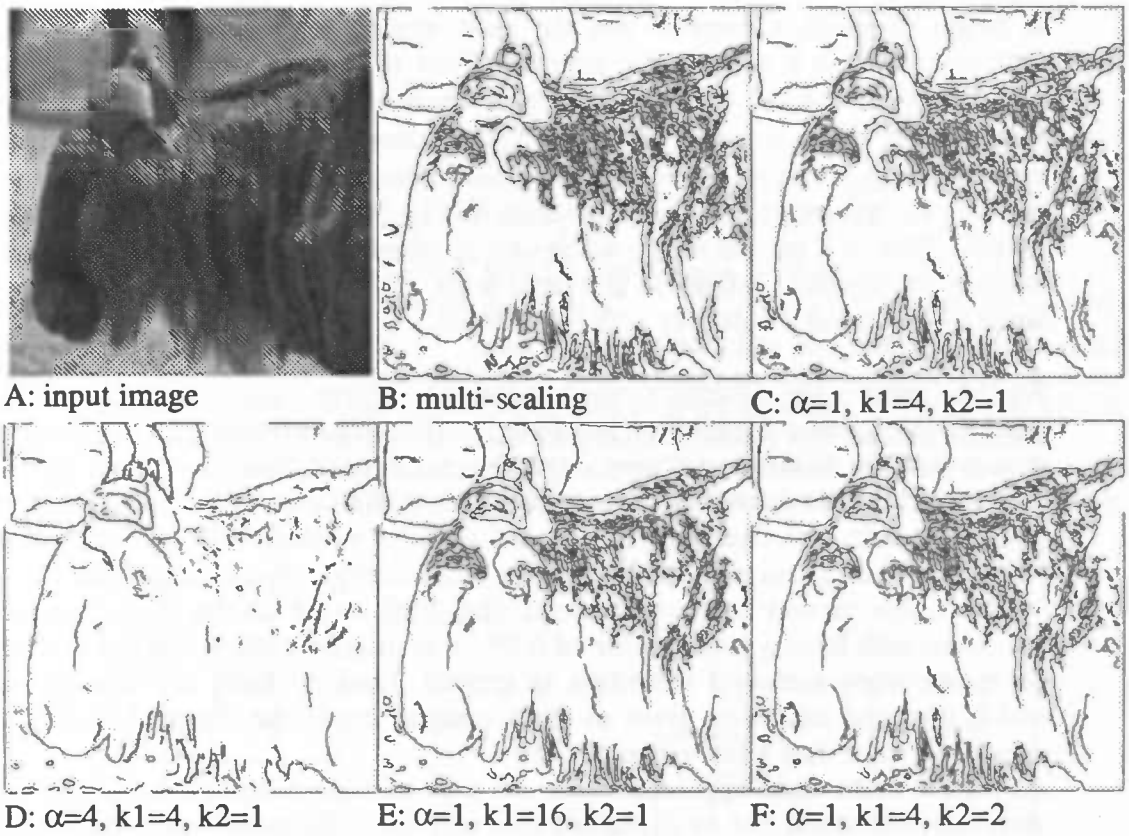


Figure 8.4: The effect of surround inhibition. *A*: Input image, *B*: multi-scaling without surround inhibition, C to *F* included: algorithm from figure 8.3 applied to the input image with different values for α k1and k2.

At last it is interesting to know what the influence of surround inhibition is on the performance of a contour detector and if in combination with surround inhibition, the multi-scaling approach from the previous chapter yields a better performance than a single scale with surround inhibition. To answer these questions, I have used surround inhibition with the Canny operator using the same σ's and thresholds as in the

previous chapter: the σ's for the single scale approach are 1, 2 and 4 and for the multi-scale approach the collection {1,2,4}. The high thresholds used are 0.1, 0.2, 0.3, 0.4, 0.5 and 0.6. The low threshold is always 0.04. For α the values 2,4, 6, and 8 are used. As can be seen in figure 8.4, the inhibition effect depends mostly on α, and little on *k1* and *k2*. To minimize the amount of different parameter combinations to test, these are kept to respectively the values 4 and 1.

In figures 8.5 to 8.8 included, the best performance achieved with and without surround inhibition for both the single and multi-scale approach are shown for a total of 4 images. Also a box-and-whisker plot is shown for the results per image.
In general one can conclude that by surround inhibition the amount of false positives decreases. This was to be expected, as surround inhibition suppresses edges close to another. Such edges are usually originating from texture and are not selected in the groundtruth. Unfortunately the amount of false negatives increases, because curly edges part of the outline of the object of interest are removed as well. This decreases the gain achieved with surround inhibition.

It might come as a surprise that the multi-scaling approach yields a worse best performance than a single scale approach even if surround inhibition is applied. In fact it is rather logical. When comparing the groundtruth image with the result of the Canny edge detector using different σ's one can observe that one specific scale mimics the groundtruth quite well for a well chosen threshold. So apparently when drawing the groundtruth, the drawer does this by looking at edges at a specific scale as well. This is a natural thing; when one is asked to draw the important edges one focuses on the whole image, so at a large scale, rather than on the details. That would simply overwhelm the drawer with information.

As the Canny edge detector's result mentioned above looks already a lot like the groundtruth, surround inhibition has little to offer. For example with the image of the gnu in figure 8.5, the best Canny edge detector's result (the third image on the first row) already gives three times less false positives than correctly detected edge pixels. Combined with the fact that the amount of false negatives increases by surround inhibition, there is no gain at all: the best result using a single scale with (the center image on the second row) and without (the third image on the first row) surround inhibition both have a performance of 0.59. It is in fact possible that the performance decreases when surround inhibition is applied. Luckely there are also images, for which surround inhibition gives an improvement, but in the images I have used this was never more than a few percent.
As for the multi-scale approach, there are a lot of false positives, so it benefits from surround inhibition, but as explained this still does not make multi-scaling a better way of contour detection than the single scale approach. Not for a single test image I used, multi-scaling outperformed the single scale approach.

Still multi-scaling has something to offer. Here a lot of scales, thresholds and inhibition factors are tested and for each image a groundtruth is drawn. In general however this is not the case and fixed parameters are chosen for all images and a groundtruth is not available. If one compares the boxes from the box-and-whisker plots one can observe that the spread in performance often decreases and the median lays higher when a multi-scale approach is taken instead of using a single scale. So

the multi-scale approch is less sensitive to the choice of parameters.

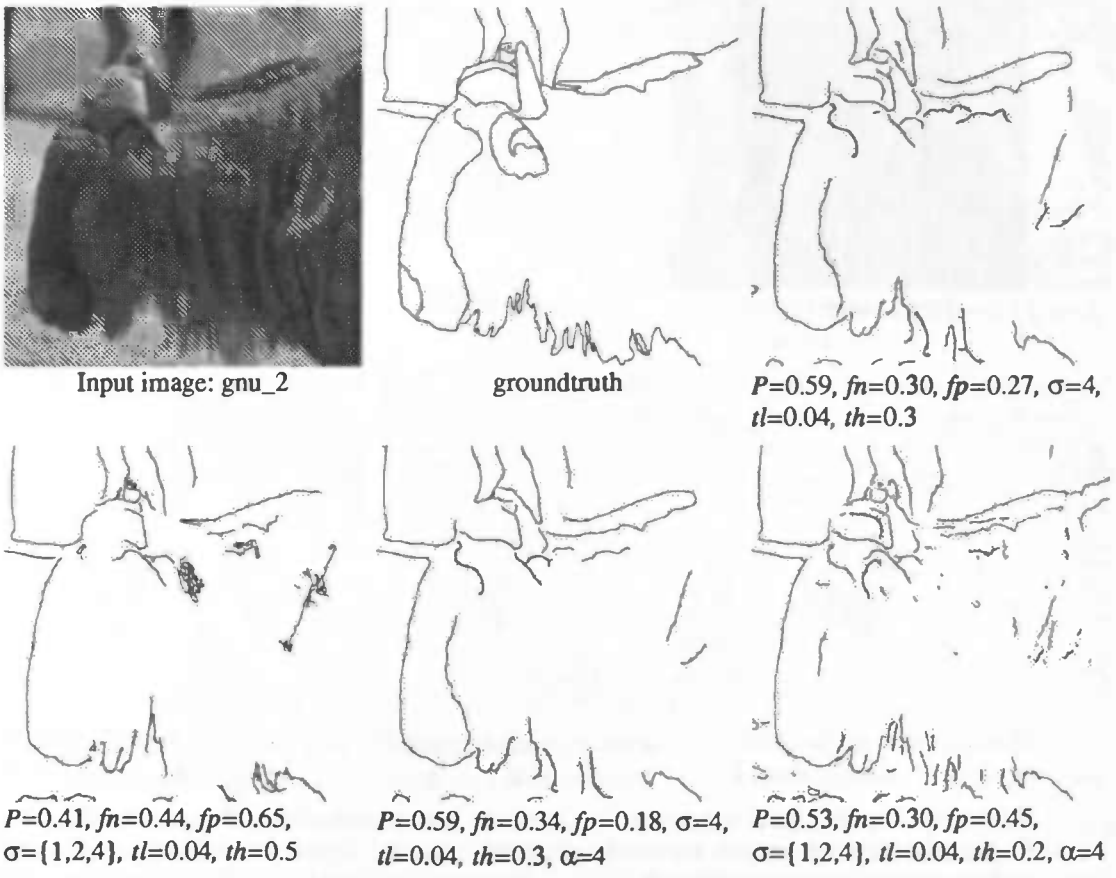
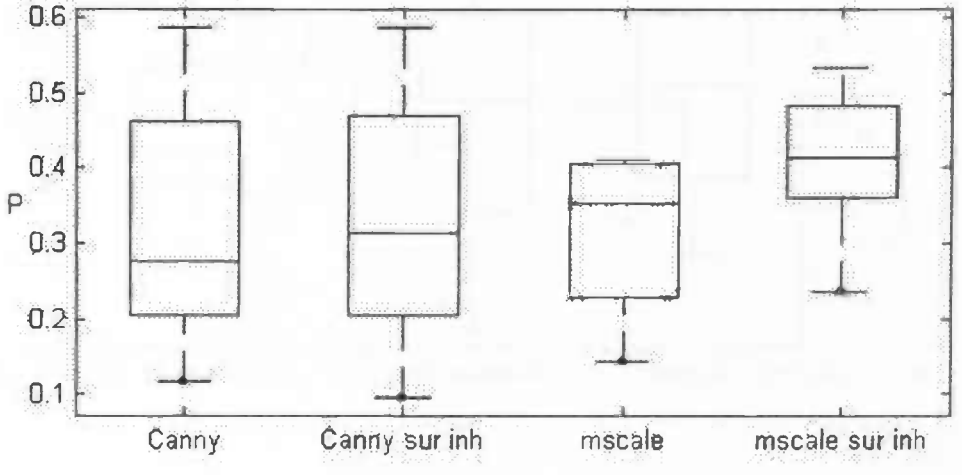


Figure 8.5: Top left: Input image gnu_2. Center: its groundtruth. Right: Best result achieved using the Canny edge detector without surround inhibition. Bottom left: Best result achieved using the multi-scaling without surround inhibition. Center: Best result achieved using the Canny edge detector with surround inhibition. Right: Best result achieved using the multi-scaling with surround inhibition
Below a box-and-whisker plot of the performance achieved for respectively the Canny edge detector, the Canny edge detector with surround inhibition, the multi-scale approach and the multi-scale approach with surround inhibition.



Input image bear_2

Groundtruth

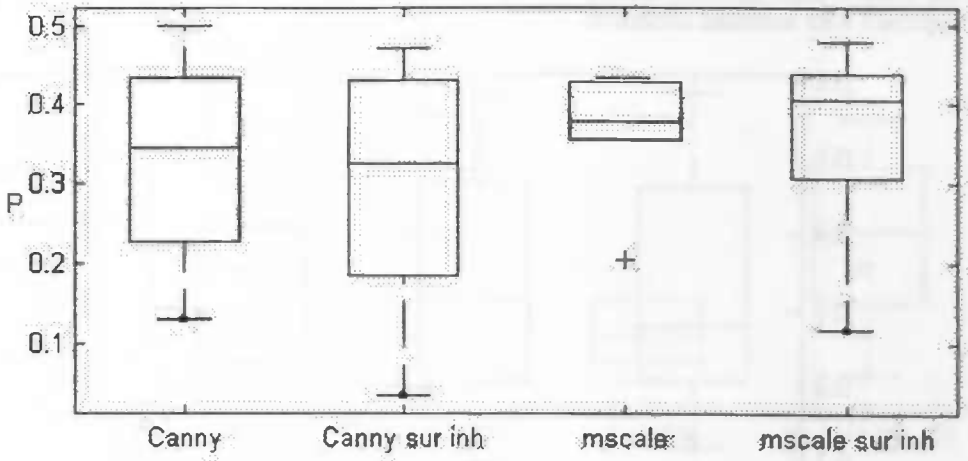
P=0.50, *fn*=0.24, *fp*=0.69, σ=4, *tl*=0.04, *th*=0.2

P=0.43, *fn*=0.47, *fp*=0.40, σ={1,2,4}, *tl*=0.04, *th*=0.4

P=0.47, *fn*=0.42, *fp*=0.39, σ=4, *tl*=0.04, *th*=0.2, α=8

P=0.48, *fn*=0.38, *fp*=0.49, σ={1,2,4}, *tl*=0.04, *th*=0.2, α=6

Figure 8.6: Top left: Input image bear_2. Center: its groundtruth. Right: Best result achieved using the Canny edge detector without surround inhibition. Bottom left: Best result achieved using the multi-scaling without surround inhibition. Center: Best result achieved using the Canny edge detector with surround inhibition. Right: Best result achieved using the multi-scaling with surround inhibition.
Below a box-and-whisker plot of the performance achieved for respectively the Canny edge detector, the Canny edge detector with surround inhibition, the multi-scale approach and the multi-scale approach with surround inhibition.



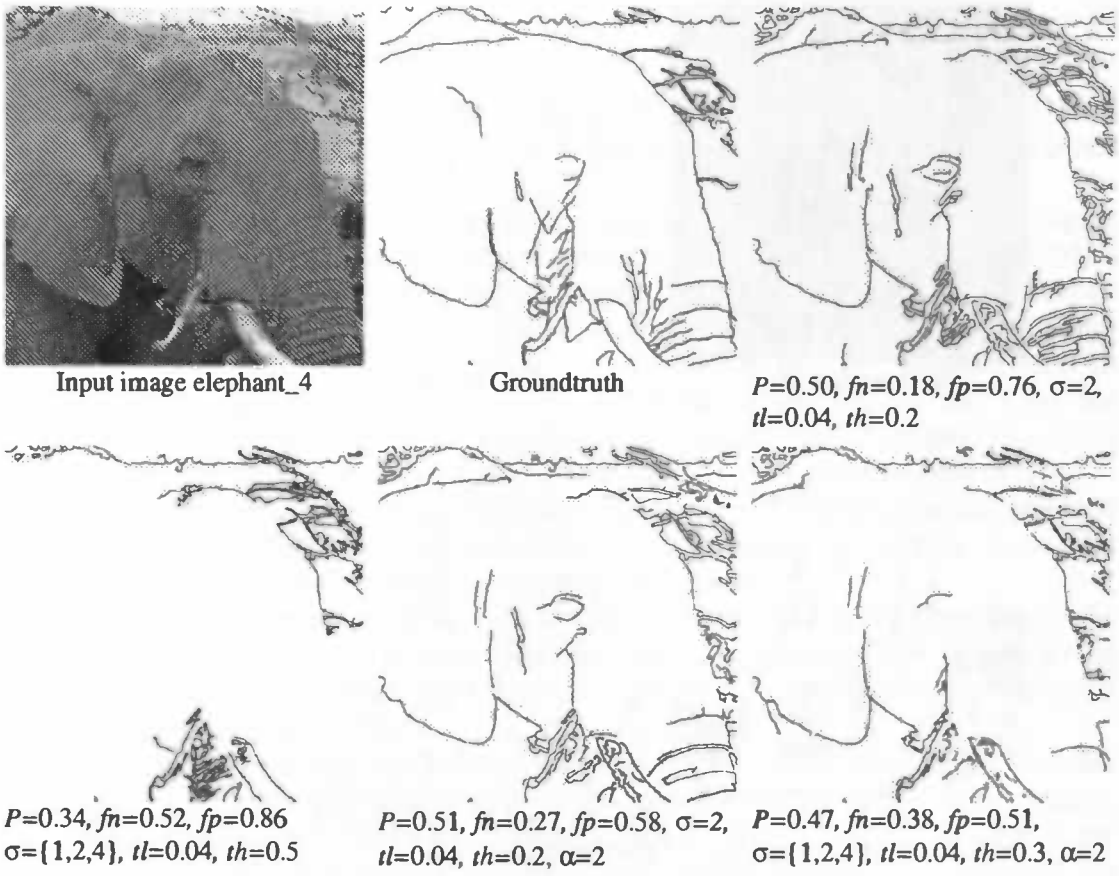
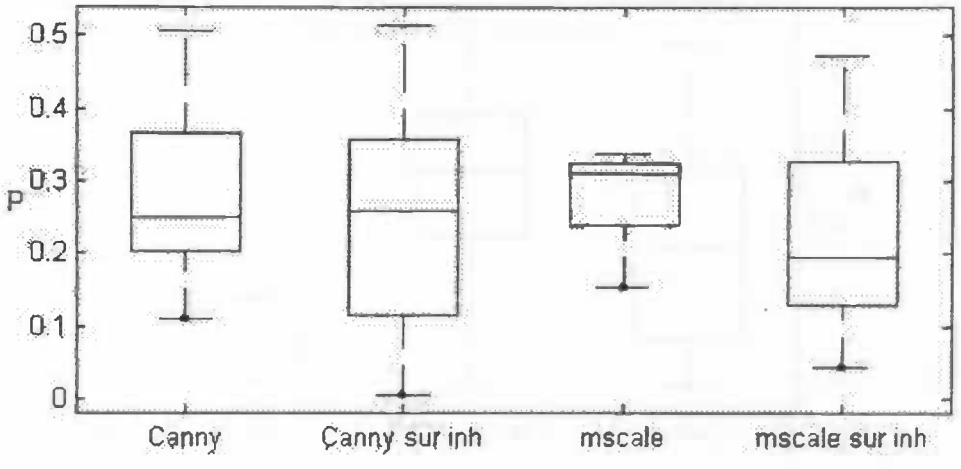
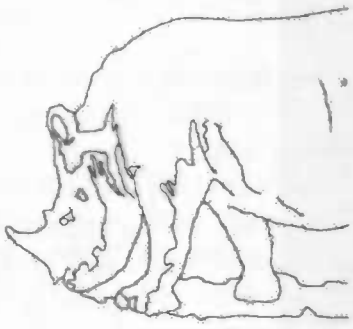


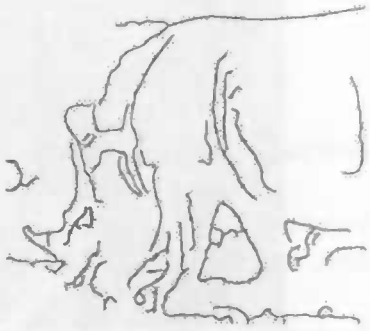
Figure 8.7: Top left: Input image elephant_4. Center: its groundtruth. Right: Best result achieved using the Canny edge detector without surround inhibition. Bottom left: Best result achieved using the multi-scaling without surround inhibition. Center: Best result achieved using the Canny edge detector with surround inhibition. Right: Best result achieved using the multi-scaling with surround inhibition
Below a box-and-whisker plot of the performance achieved for respectively the Canny edge detector, the Canny edge detector with surround inhibition, the multi-scale approach and the multi-scale approach with surround inhibition.



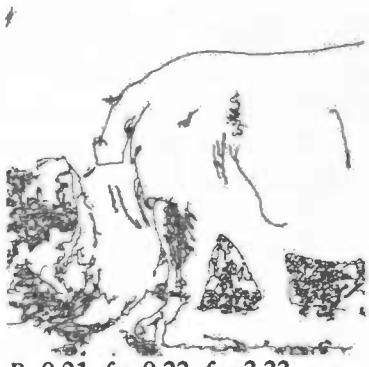
Input image rino_2



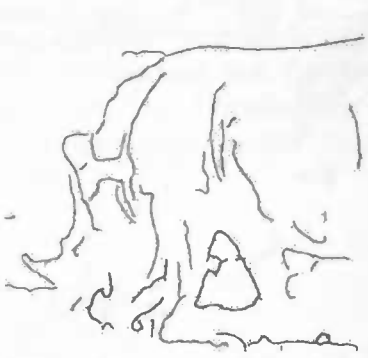
Groundtruth



P=0.54, *fn*=0.30, *fp*=0.44, σ=4, *tl*=0.04, *th*=0.5



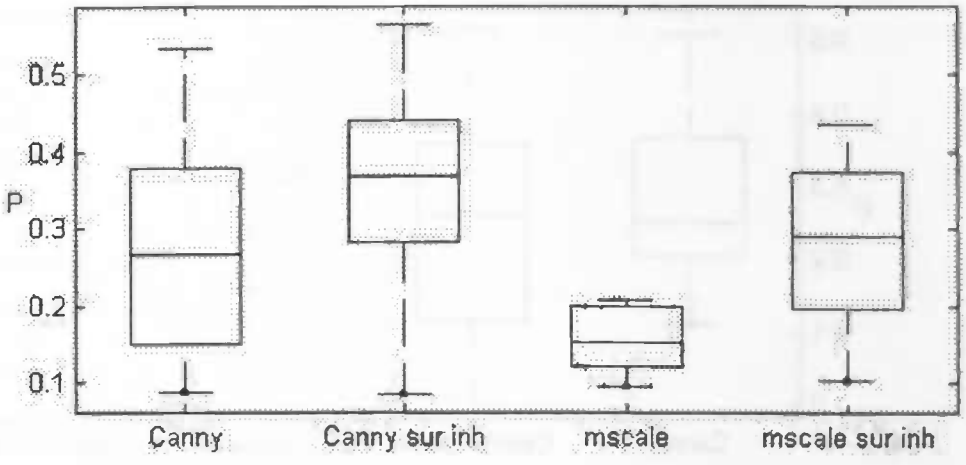
P=0.21, *fn*=0.33, *fp*=3.33 σ={1,2,4}, *tl*=0.04, *th*=0.5



P=0.57, *fn*=0.33, *fp*=0.27, σ=4, *tl*=0.04, *th*=0.3, α=8

P=0.44, *fn*=0.44, *fp*=0.49, σ={1,2,4}, *tl*=0.04, *th*=0.3, α=4

Figure 8.8: Top left: Input image rino_2. Center: its groundtruth. Right: Best result achieved using the Canny edge detector without surround inhibition. Bottom left: Best result achieved using the multi-scaling without surround inhibition. Center: Best result achieved using the Canny edge detector with surround inhibition. Right: Best result achieved using the multi-scaling with surround inhibition
Below a box-and-whisker plot of the performance achieved for respectively the Canny edge detector, the Canny edge detector with surround inhibition, the multi-scale approach and the multi-scale approach with surround inhibition.



9. Conclusions

In this thesis it is shown that the creation of a contour detector with a high performance is no trivial task. The method that gives the best performance depends greatly on the kind of images used.
First of all it is important whether the input image is an artificial drawing or a digital photograph. For the first class of images the multi-scaling approach clearly yields a higher performance than a single-scale approach. If however digital photographs are used, with the single scale approach a higher performance can be achieved than using the multi-scale technique. This is mainly caused by the fact that humans tend to interpret edges at a single scale as well. As hand drawn images are used for comparison this is rather logical. Still multi-scaling can be useful as normally a fixed set of parameters is chosen with a contour detector. As the spread in performance is lower and the median higher using the multi-scale approach, the multi-scale approach is less sensitive to the choice of parameters and the same set can be used with different images, while still achieving a good performance.
The influence of surround inhibition on the performance also depends much on the kind of images used. Images in which texture edges are present at the same scale as the object contours and which have high strength benefit from surround inhibition (look for example at figure 8.8).
Unfortunately, although the best contour detection results achieved using the methods described in the thesis are recognizable as the object(s) shown in the input images, still quite a few important edges are missing. Often parts of the outline of objects are not detected and the same holds for facial characteristics like the eyes, ears and the nose of animals. Clearly the gradient of Gaussian is useful in edge detection, but in order to create a good contour detector more information is needed. Here other things are important as well like knowledge about how objects look like in reality. A human knows that if some important characteristic is not very clear in the picture or not present at all like a part of the outline of a face, he still knows it is there and he would complete the image by using his experience. The algorithms described here do not take into account such knowledge and it remains a challenge to come up with a method to make a computer clear what is trivial to us.

10. References

[1]N. Efford, "Digital image processing, a practical introduction using Java", ISBN 0-201-59623-7, Addison Wesley, 2000

[2]M. Sonka, V. Hlavac and R. Boyle, "Image processing, analysis and machine vision, second edition", ISBN 0-534-95393-X, PWS Publishing, 1999.

[3]H.D. Tagare and R.J.P. de Figueiredo, "On The Localization Performance Measure And Optimal Edge Detection", *IEEE Transactions On Pattern Analysis And Machine Vision*, vol. 12, no12, December 1990.

[4]C. Grigorescu, N. Petkov and M. Westenberg, "Performance enhancement of contour detectors by surround inhibition", *Zakopane* 25-29, pg. 313-318, sept. 2002.

[5]J.F. Canny, "A computational approach to edge detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, Nov 1986.

[6]K.L. Boyer and S. Sarkar, "On The Localization Performance Measure And Optimal Edge Detection", *IEEE Transactions On Pattern Analysis And Machine Vision*, vol. 16, no1, January 1994.

11. Appendix A: Matlab source

```
--------------------------------------------------------------------------------
                                  Contents.m
--------------------------------------------------------------------------------

% Contour Detection Toolbox
% by Reinco Hof
%
% Contour Detection Toolbox GUI (Graphical User Interface)
%   CONTOURDEMO                   - Contour detection demo with GUI
%
% Preprocessing: image smoothing
%   GAUSSIAN                      - Gaussian smoothing of an image
%   GAUSSIAN2D                    - two demensional Gaussian function
%   GAUSSIANCONVOLUTIONKERNEL2D   - 2D Gaussian convolution kernel
%
% Edge enhancement
%   GRADIENTOFGAUSSIAN            - Gaussian smooths an image and returns the gradient
%   DERIVATIVEOF2DGAUSSIANKERNEL  - constructs a dG filter
%   MULTISCALE                    - gradient computation using multiple scales
%   MKMULTISCALEIMG               - creates a test image for multiscaling
%
% Edge thinning
%   THINNENEDGESLINEAR            - non-maximum suppression using lineair interpolation
%                                   scheme
%   THINNENEDGESSIMPLE            - edge thinning using non-maximum suppression
%
% Binarization
%   THRESHOLD                     - thresholds an image
%   HYSTERESISTHRESHOLD           - thresholds an image with hysteresis
%
% Complete edge detectors
%   CANNY                         - canny edge detector
%
% Edge selection
%   SURROUNDINHIBITION            - suppresses edges close to others in a gradient
%                                   magnitude image
%   DOGCONVOLUTIONKERNEL2D        - constructs a DoG convolution kernel
%
% Complete contour detectors
%   DGOP                          - contour detection using the dG filter
%
% Performance measurement
%   RHPERFORMANCE                 - measures the performance of a contour detector
%
% Basic imaging functions
%   RHLOADIMAGE                   - loads an image from a file
%   SAVEIMG                       - stores an image in a file
%   RHIMAGE                       - displays an image
%   RHMKIMGVIEWABLE               - transforms an image into one that can be displayed
%   TOINTENSITYIMG                - translates an image to an intensity image
%   SCALELINEAR                   - scales the values of an image to range [0, 1]
%   CONVOLVE                      - applies convolution
%   NORMALIZE                     - normalizes a matrix with the L1-norm
%   GETGRADIENTMAGNITUDE          - finds the gradient magnitude
%
% Functions dealing with the special type of image variables used in this toolbox
%   PIMAGE                        - creation of a variable of type PImage.
%   TIMAGE                        - creates a TImage structure
%   ISTIMAGE                      - true for TImages
%   FREEPIMAGE                    - give up memory used by a PIMAGE
%
%
% Helper functions
%   RHFIX                         - floor that takes computational errors into account
%   RHSIGN                        - sign function insensitive to computational noise
%   MKVALID                       - maps nonfinite values to another, finite value
%   RHIMGFFT                      - updates the FFT of an image
%   REFLECT                       - enlarges an image by reflecting it at its borders
%   MAXIMG                        - the pixelwise maximum of two images
%   MKLOGLIST                     - creates a list with values on a logaritmic scale
```

contourdemo.m

```
function contourdemo(varargin)
% CONTOURDEMO Contour detection demo with GUI
%  CONTOURDEMO demonstrates contour detection using a graphical
%    user interface. On the left the input image and on the right
%    the contour detector's result is shown. A click on these images
%    shows them fullsize in a seperate window. Beneath the image three
%    panels are present. In the left the input image can be selected in
%    the combobox or you can supply your own image with the '...' button.
%    In the center panel the contour detection options can be set.
%    Information about each button in the GUI is shown when the mouse is
%    moved over it in a tooltip. For detailed information about parameters, read the
%    help information about the DGOP function.
%    When you have selected the appropriate parameters, click the 'Apply' button
%    in the right panel. Depending on the complexity of the operation, the computation
%    might take 10 seconds to 3 minutes. When finished, you can save the result with
%    the 'Save' button.
%
% See also: DGOP

if (nargin == 0)
 % code executed when function is called by user

 % ------------------------ init variables ------------------------
 clear d; % d is the structure storing handles, variables of the figure
 d.sigma = 1.0; % SD used by the (derivative of) Gaussian filter
 d.sigmalow = 1.0; % min and max sigma used with multiscaling
 d.sigmahigh = 8.0;
 d.filterMethod = 'gog'; % filter to apply to the input image (currently only 'gog')
 d.thresholdLow = 10; % low threshold for hysteresis thresholding (0..255)
 d.threshold = 32; % threshold value (0..255)
 d.thinnen = 0; % 0 = no edge thinning, >0 applies edge thinning
 d.brightness = 0; % no brightness enhancement (0..2)
 d.thresholdMethod = 0; % no thresholding
 d.alpha = 2.0; d.k1 = 4; d.k2 = 1; % alpha, k1, k2 to use with surround inhibition
 d.filename = 'contourdemo/lena.jpg'; % default input image
 d.Lx = ''; d.Ly = ''; % gradient of the image (unknown)
 d.winners = ''; % sigma used per pixel (unknown)
 d.gradientcomputation = 'single-scale'; % method of gradient computation


 % create a resizable figure window without menubar
 d.hfig = figure('MenuBar', 'none','Position',[100,100,600,600],...
                                        'NumberTitle', 'off', 'Name', 'Contour detection demo',...
                                        'Resize', 'off',...
                                        'WindowButtonUpFcn', mkCallBack('mouseClick'), ...
          'CloseRequestFcn', mkCallBack('closeFigure'));
 dims = get(d.hfig, 'Position'); % get figure dimensions
 w = dims(3); % (w,h) = width x height of the figure
 h = dims(4);
 w2 = fix(w/2); % (w2,h2) is the figure's center
 h2 = fix(h/2);

 % GUI parameters
 d.bgColor = [0.8, 0.8, 0.8]; % background color to use
 d.Indent = [5,5]; % indent to use with controls in panels
 d.ImgLabelsPanelHeight = 20; % height of the image labels panel
 d.ButtonsPanelHeight = 280; % height of the panel with the buttons
 d.InputPanelSize = [200, 80]; % size of the input panel
 d.OutputPanelSize = [100, 110]; % size of the output panel



% ---------------- gui creation ----------------------------
 % create the gui contents (axes, buttons, etc). Below is an ascii
 % art of the gui's layout. Controls drawn at the same location are
 % denoted by ControlA/ControlB.
 %
```

```
%                    ---------------------------------
%ImgLabelsPanel     | InputTxt          OutputTxt     |
%                   | ------------    ---------------  |
%                   ||            | |               ||
%ImgsPanel          || Inputimage  | | Outputimage   ||
%                   ||            | |               ||
%                   ||            | |               ||
%                   | ------------    ---------------  |
%                   |  _______  _____________  _______ |
%                   | |       ||             ||       ||
%                   | |       ||             ||       ||
%ButtonsPanel       | |       ||             ||       ||
%                   | |_______||             ||_______||
%                   |         |_____________|         |
%                    ---------------------------------
%                   InputPanel OptionsPanel OutputPanel
%
%
% InputPanel: --------------------------
%             | SelectImgTxt             |
%             | InputImgCombo InputImgBtn|
%              --------------------------
%
% OptionsPanel:-------------------------------------------------------------------
%             |  ---------------------------------------------------------------  |
%             | | GradientCalcMethodCombo SigmaEdit/SigmaLowEdit SigmaHighEdit | |
% GradientComputationPanel
%             |  ---------------------------------------------------------------  |
%             |  ---------------------------------------------------------------  |
%             | | ThinningChk                                                   | |
% ThinningPanel
%             |  ---------------------------------------------------------------  |
%             |  ---------------------------------------------------------------  |
%             | | SurroundInhibitionChk AlphaEdit K1Edit K2Edit                 | |
% EdgeSelectionPanel
%             |  ---------------------------------------------------------------  |
%             |  ---------------------------------------------------------------  |
%             | | ThresholdCombo BrightnessCombo/ThresholdLowEdit ThresholdEdit| |
% ThresholdPanel
%             |  ---------------------------------------------------------------  |
%              -------------------------------------------------------------------
%
% OutputPanel:--------------------------
%             | EvaluateBtn              |
%             | SaveBtn                  |
%             | Help                     |
%              --------------------------
```

```
% Image labels panel:
d.hImgLabelsPanel = addControl(d, 'Style', 'Panel', 'Position', [0, 0, w, d.ImgLabelsPanelHeight]);
d.hInputTxt = addControl(d, 'Style', 'Text', 'Frame', d.hImgLabelsPanel, 'Position', [0, 0, w2, d.ImgLabelsPanelHeight],...
                                                    'String', 'Input Image');
d.hOutputTxt = addControl(d, 'Style', 'Text', 'Frame', d.hImgLabelsPanel, 'Position', [w2, 0, w2, d.ImgLabelsPanelHeight],...
                                                    'String', 'Output Image');

% Images panel:
ImgsPanelHeight = max(0, h - d.ImgLabelsPanelHeight - d.ButtonsPanelHeight - 2*d.Indent(1));
d.hImgsPanel = addControl(d, 'Style', 'Panel', 'Position', [0, d.ImgLabelsPanelHeight, w, ImgsPanelHeight]);
d.hInputimage = addControl(d, 'Style', 'Axes', 'Frame', d.hImgsPanel, 'Position', [0, 0, w2, ImgsPanelHeight], 'Visible', 'off');
d.hOutputimage = addControl(d, 'Style', 'Axes', 'Frame', d.hImgsPanel, 'Position', [w2, 0, w2, ImgsPanelHeight], 'Visible', 'off');

% Buttons panel:
d.hButtonsPanel = addControl(d, 'Style', 'Panel', 'Position', {'beneath', d.hImgsPanel, w, d.ButtonsPanelHeight});
ips = d.InputPanelSize;
ops = d.OutputPanelSize;
%Input panel:
 d.hInputPanel = addControl(d, 'Style', 'Frame', 'Frame', d.hButtonsPanel, 'Position', [d.Indent(1), 0, ips(1), ips(2)]);
 d.hSelectImgTxt = addControl(d, 'Style', 'Text', 'Frame', d.hInputPanel, 'Position', [d.Indent(1), d.Indent(2), 150, 20],...
                                                    'String', 'Select an image', 'HorizontalAlign', 'left');
 d.hInputImgCombo = addControl(d, 'Style', 'popupmenu', 'Position', {'beneath', d.hSelectImgTxt, 150, 20},...
                                    'String', 'lena 512x512x256|gnu 512x512x256|blocks 512x512x256',...
                                    'Callback', mkCallBack('InputImgComboClk'));
 d.hInputImgBtn = addControl(d, 'Position', {'rightof', d.hInputImgCombo, 30, 20},...
                                    'String', '...', 'CallBack', mkCallBack('InputImgBtnClk'));
%Options panel:
 opw = max(0, w-ips(1)-ops(1)-d.Indent(1)*3);
 maxpw = opw - d.Indent(1)*2; % maximum width a control can have in the options panel
 d.hOptionsPanel = addControl(d, 'Style', 'Frame', 'Position', {'rightof', d.hInputPanel, opw, d.ButtonsPanelHeight});
 % Gradient computation panel:
  d.hGradientComputationPanel = addControl(d, 'Style', 'Frame', 'Frame', d.hOptionsPanel, 'Position',...
                                                    [d.Indent(1),d.Indent(2),maxpw,50],...
```

```
                                                    'String', 'Gradient computation');
  if ((strcmp(d.gradientcomputation, 'multi-scale')))
                    multi = 'on';
                    single = 'off';
  else
                    multi = 'off';
                    single = 'on';
  end
  p = d.hGradientComputationPanel;
  d.hGradientCalcMethodCombo = addControl(d,'Style','popupmenu', 'Position',{'beneath',getCaption(p),100,20},...
                                                    'String', 'single-scale|multi-scale', 'Callback',...
                                                    mkCallBack('GradientCalcMethodComboClk'));
  d.hSigmaEdit = addControl(d,'Style','edit', 'Position',{'rightof',d.hGradientCalcMethodCombo,50,20}, 'Visible', single,...
                                    'String', num2str(d.sigma), 'Callback', mkCallBack('sigmaEditEnterPressed'));
  d.hSigmaLowEdit = addControl(d,'Style','edit', 'Position',{'rightof',d.hGradientCalcMethodCombo,50,20}, 'Visible', multi,...
                                    'String', num2str(d.sigmalow), 'Callback', mkCallBack('sigmaLowEditEnterPressed'));
  d.hSigmaHighEdit = addControl(d,'Style','edit', 'Position',{'rightof',d.hSigmaLowEdit,50,20}, 'Visible', multi,...
                                    'String', num2str(d.sigmahigh), 'Callback', mkCallBack('sigmaHighEditEnterPressed'));
% Edge thinning panel:
  d.hThinningPanel = addControl(d, 'Style', 'Frame', 'Position', {'beneath',d.hGradientComputationPanel,maxpw,50},...
                                    'String', 'Edge thinning');
  d.hThinningChk = addControl(d, 'Style', 'check', 'Position', {'beneath',getCaption(d.hThinningPanel),70,20},...
                                    'String', 'Apply', 'Value', d.thinnen, 'Callback', mkCallBack('thinningChkClk'));
% edge selection panel
  d.hEdgeSelectionPanel = addControl(d, 'Style', 'Frame', 'Position', {'beneath',d.hThinningPanel,maxpw,50},...
                                    'String', 'Surround inhibition');
  d.hSurroundInhibitionChk = addControl(d, 'Style', 'check', 'Position', {'beneath',getCaption(d.hEdgeSelectionPanel),70,20},...
                                    'String', 'Apply');
  d.hAlphaEdit = addControl(d,'Style','edit', 'Position',{'rightof',d.hSurroundInhibitionChk,50,20},...
                                    'String', num2str(d.alpha), 'Callback', mkCallBack('alphaEditEnterPressed'));
  d.hK1Edit = addControl(d,'Style','edit', 'Position',{'rightof',d.hAlphaEdit,50,20},...
                                    'String', num2str(d.k1), 'Callback', mkCallBack('k1EditEnterPressed'));
  d.hK2Edit = addControl(d,'Style','edit', 'Position',{'rightof',d.hK1Edit,50,20},...
                                    'String', num2str(d.k2), 'Callback', mkCallBack('k2EditEnterPressed'));
% threshold panel
  d.hThresholdPanel = addControl(d, 'Style', 'Frame', 'Position', {'beneath',d.hEdgeSelectionPanel,maxpw,50},...
                                    'String', 'Binarization');
  d.hThresholdCombo = addControl(d, 'Style', 'popupmenu', 'Position', {'beneath', getCaption(d.hThresholdPanel), 120, 20},...
                                    'String', 'no thresholding|thresholding|hysteresis',...
                                    'Callback', mkCallBack('thresholdComboChanged'));
  d.hThresholdLowEdit = addControl(d,'Style','edit', 'Position',{'rightof',d.hThresholdCombo,50,20},'Visible','off',...
                                    'String', num2str(d.thresholdLow), 'Callback',...
                                    mkCallBack('thresholdLowEditEnterPressed'));
  d.hThresholdEdit = addControl(d,'Style','edit', 'Position',{'rightof',d.hThresholdLowEdit,50,20},'Visible','off',...
                                    'String', num2str(d.threshold), 'Callback', mkCallBack('thresholdEditEnterPressed'));
  d.hBrightnessCombo = addControl(d, 'Style', 'popupmenu', 'Position', {'rightof', d.hThresholdCombo, 100, 20},...
                                    'String', 'dark|medium|bright','Callback', mkCallBack('brightnessChanged'));
%Output panel:
  d.hOutputPanel = addControl(d, 'Style', 'Frame', 'Position', {'rightof', d.hOptionsPanel, ops(1), ops(2)});
  d.hEvaluateBtn = addControl(d, 'Frame', d.hOutputPanel, 'Position', [d.Indent(1), d.Indent(2),...
                                    getMaxObjWidth(d,d.hOutputPanel), 30],...
                                    'String', 'Apply', 'BackgroundColor', 'green',...
                                    'CallBack', mkCallBack('evaluateBtnClk'));
  d.hSaveBtn = addControl(d, 'Position', {'beneath',d.hEvaluateBtn,getMaxObjWidth(d,d.hOutputPanel), 30},...
                                    'String', 'Save', 'CallBack', mkCallBack('SaveBtnClk'));
  d.hHelpBtn = addControl(d, 'Position', {'beneath',d.hSaveBtn,getMaxObjWidth(d,d.hOutputPanel), 30},...
                                    'String', 'Help', 'CallBack', mkCallBack('HelpBtnClk'));


% Set the tooltips per GUI component:
d = setGUITooltips(d);

%---------- load and set input and output image in the gui ----------

% load input image
d.img = PImage(rhloadimage(d.filename));

% show input image and hide its axes
rhimage(TImage(d.img), d.hInputimage);

% create empty output image (ie image with same color as the background)
outputImagePos = get(d.hOutputimage, 'Position');
for i=1:3,
  d.outimg(:, :, i) = ones(outputImagePos(4), outputImagePos(3)) * d.bgColor(i);
end
```

```
  d.outimg = TImage(d.outimg);

  % store the record structure d with the figure
  set(d.hfig, 'UserData', d);

else
  % code executed on callback. use form: contourdemo(
  % 'functionName', arg1, ...). this will execute d = functionName(arg1,
  % ..., d); where functionName is a contourdemo local
  % function and d is the record structure stored with the figure
  d = get(gcf, 'UserData'); % retreive data stored with the figure
  if (~isempty(d)) % always true, except if a callback occurs
                % during initialization phase. We must check for
                % this case, because only after the
                % initialisation all variables in d are
                % initialized.
    % perform the function call. The functions arguments are
    % collected in variable t, which is a structure array.
    clear t;
    for i=2:length(varargin), % collect function arguments
      t{i-1} = (varargin{i});
    end
    t{length(varargin)} = d;

    d = feval(varargin{1}, t{:}); % execute function
    if (~isempty(d))
      % d is empty if the figure is deleted, so then d.hfig is invalid
      set(d.hfig, 'UserData', d); % execute function and store data with figure
    end
  end
end


function result=setGUITooltips(d)
% set the tooltip texts of each GUI component

set(d.hInputImgCombo, 'TooltipString', 'Select an input image');
set(d.hInputImgBtn, 'TooltipString', 'Select an image file as input image');
set(d.hGradientCalcMethodCombo, 'TooltipString', 'Gradient computation method');
set(d.hSigmaEdit, 'TooltipString', 'Standard deviation (0,32] to use in the gradient computation');
set(d.hSigmaLowEdit, 'TooltipString', 'Minimum standard deviation (0,32] to use in the gradient computation');
set(d.hSigmaHighEdit, 'TooltipString', 'Maximum standard deviation (0,32] to use in the gradient computation');
set(d.hThinningChk, 'TooltipString', 'Thinnen the edges to one pixel wide');
set(d.hSurroundInhibitionChk, 'TooltipString', 'Surround inhibition. In the 3 edit boxes alpha, k1 and k2 can be entered');
set(d.hAlphaEdit, 'TooltipString', 'Suppression factor alpha [0,Inf) to use with surround inhibition');
set(d.hK1Edit, 'TooltipString', 'Factor k1 [0,Inf) to use with surround inhibition');
set(d.hK2Edit, 'TooltipString', 'Factor k2 (0,k1) to use with surround inhibition');
set(d.hThresholdCombo, 'TooltipString', 'Thresholding method to use');
set(d.hBrightnessCombo, 'TooltipString', 'Enhance the brightness of the output image');
set(d.hThresholdLowEdit, 'TooltipString', 'Value of low threshold TLOW [0,255] to use with hysteresis thresholding');
d.ThresholdEditTooltipTxt = 'Threshold value [0,255]';
d.ThresholdHighEditTooltipTxt = 'Value of high threshold THIGH [0,255] to use with hysteresis thresholding';
set(d.hThresholdEdit, 'TooltipString', d.ThresholdHighEditTooltipTxt);
set(d.hEvaluateBtn, 'TooltipString', 'Apply the specified contour detector to the input image');
set(d.hSaveBtn, 'TooltipString', 'Save the contour detector''s result as a .jpg or .gif file');
set(d.hHelpBtn, 'TooltipString', 'Opens the contour demo help file in your browser');
result = d;

function result = evaluateBtnClk(d)
% called when the evaluate button is clicked

% apply the filter the user has selected to d.fimg
switch (d.filterMethod)
 case 'gog'
  % apply the dG filter with SD = d.sigma

  % get sigmas to use for gradient computation
  if (strcmp(d.gradientcomputation, 'single-scale'))
    sigmas = d.sigma;
    if isstr(d.Lx)
      % gradient needs to be recomputed
      [d.Lx, d.Ly]  = gradientOfGaussian(d.img, sigmas, 1);
      d.winners = ones(size(d.Lx.img));
    end
```

```
  % thinnen edges:
  if (d.thinnen > 0)
    M = thinnenEdgesSimple(d.Lx, d.Ly);
  else
    M = getGradientMagnitude(d.Lx, d.Ly);
  end
 else
  sigmas = mkLogList(d.sigmalow, d.sigmahigh, sqrt(2));
  if isstr(d.Lx)
   % gradient needs to be recomputed
   [d.Lx, d.Ly, d.winners] = multiscale(d.img, sigmas);
  end
  M = getGradientMagnitude(d.Lx, d.Ly);
 end;

 % apply surround inhibition if corresponding checkbox checked and parameters are correct:
 if (d.k1 > d.k2) & (d.k2 > 0) & get(d.hSurroundInhibitionChk, 'value')
  Mp = PImage(M);
  M = surroundInhibition(Mp, d.alpha, d.k1, d.k2, sigmas, d.winners);
  freePImage(Mp);
 end

 % apply thresholding:
 if (d.thresholdMethod == 1)
  d.outimg = threshold(M, d.threshold / 255.0);
 elseif (d.thresholdMethod == 2)
  d.outimg = hysteresisThreshold(M, d.thresholdLow / 255.0, d.threshold / 255.0);
 else
  d.outimg = toIntensityImg(scaleLinear(M), d.brightness);;
 end
end

% change the output image to the calculated image d.outimg
rhimage(d.outimg, d.hOutputimage);

% hide axes
set(d.hOutputimage, 'Visible', 'off');

% return the modified structure d
result = d;

function result = mkCallBack(functionname)
% create a string of form 'contourdemo("", "functionname")'
result = ['contourdemo("', functionname, '")'];

function result = coord(oleft, otop, owidth, oheight, h)
% translates coordinate system with (0,0) at topleft to coordinate
% system with (0,0) at bottomleft
result(1) = oleft;
result(2) = h - (otop + oheight);
result(3) = owidth;
result(4) = oheight;

function result = getpos(hcontrol)
% get position of given control
fig = get(hcontrol, 'Parent');
figpos = get(fig, 'Position');
cpos = get(hcontrol, 'Position');
result = cpos;
result(2) = figpos(4)-cpos(2)-cpos(4);

function result = setpos(d, hcontrol, a, b)
% calculate the new position of a control
% setpos(d, hcontrol, pos):
%   pos can be:
%   1) [top, left]
%   2) [top, left, width, height]
%   3) {'leftof'|'rightof'|'beneath'|'above', controlhandle}
%   4) {'leftof'|'rightof'|'beneath'|'above', controlhandle, width, height}
% setpos(d, hcontrol, hframe, pos):
%   set position relative to hframe. pos can be:
%   1) [top, left]
%   2) [top, left, width, height]
if (nargin == 3)
 pos = a;
```

```
else
 pos = b;
 hframe = a;
end;

% get new width and height of the control:
if length(pos) == 2
 % only 2 params -> use old width (ie first and third form of pos)
 oldpos = getpos(hcontrol); % get old location of the control
 w = oldpos(3);
 h = oldpos(4);
else
 % use new width
 w = pos(3);
 h = pos(4);
end
if (iscell(pos))
 % position is left of, above, ... another control
 cpos = getpos(pos{2}); % position of control with given controlhandle
 w = w{1}; h = h{1};
 switch lower(pos{1})
  case 'leftof'
   pos = [cpos(1) - w - d.Indent(1), cpos(2), w, h];
  case 'rightof'
   pos = [cpos(1) + cpos(3) + d.Indent(1), cpos(2), w, h];
  case 'above'
   pos = [cpos(1), cpos(2) - h - d.Indent(2), w, h];
  case 'beneath'
   pos = [cpos(1), cpos(2) + cpos(4) + d.Indent(2), w, h];
  otherwise
   error('Illegal position format');
 end
else
 % position is absolute
 pos = [pos(1), pos(2), w, h];
end

if (nargin == 3)
 % pos is absolute
 figpos = get(d.hfig, 'Position'); % get position of the figure d.hfig
 result = coord(pos(1), pos(2), pos(3), pos(4), figpos(4));
else
 % pos is relative to another frame
 result = framecoord(pos(1), pos(2), pos(3), pos(4), hframe);
end


function result = framecoord(oleft, otop, owidth, oheight, frame)
% coordinates relative to given frame to figure coordinates
fig = get(frame, 'Parent');
figpos = get(fig, 'Position');
framepos = get(frame, 'Position');
result(1) = oleft + framepos(1);
result(2) = (figpos(4) - (otop + oheight + (figpos(4)-framepos(2)-framepos(4))));
result(3) = owidth;
result(4) = oheight;

function result = getCaption(hFrame)
% get handle of the caption of the frame (a text object)
result = get(hFrame, 'UserData');

function result = getMaxObjWidth(d,h)
% get the maximal width a child object can have in uicontrol with given handle (ie uicontrol width minus the indent)
pos = get(h, 'Position');
result = pos(3) - d.Indent(1)*2;

function result = addControl(d, varargin)
% add a uicontrol to the figure d.hfig. Rest of the parameters equal that
% of the uicontrol function. There is one extra parameter: 'Frame'. If present
% then the Position is relative to the given frame. If no background color is
% specified d.bgColor is used.
% addControl accepts two extra styles:
%  - 'Panel', this is a frame without border.
%  - 'Axes', creates an axes. For arguments see the axes function
% You can place the new control next to another one: 'Position', {'leftof'|'rightof'|'beneath'|'above', hControl, width, height}
```

```
%Units is always pixels.

% place cell arrays in varargin in a cell array, ie: {1, 2, {3, 4}} -> {1, 2, {{3, 4}}}. function struct demands this
for i=1:length(varargin),
 if iscell(varargin{i})
  varargin{i} = {varargin{i}};
 end
end

r = struct(varargin{:}); % get the varargs as structure

% get the relative position where the new control should come:
if (isfield(r, 'Position'))
 % allowed position formats: [left, top, width, height] and {width, height, 'leftof'|'rightof'|'beneath'|'above', controlhandle}
 if isfield(r, 'Frame')
  % field frame exists, so position is frame relative }
  r.Position = setpos(d, 0, r.Frame, r.Position);
  % remove the Frame field from r }
  r = rmfield(r, 'Frame');
 else
  r.Position = setpos(d, 0, r.Position);
 end
end

if (~isfield(r, 'BackgroundColor'))
 % if no background color specified, use d.bgColor
 r.BackgroundColor = d.bgColor;
end

% Handle extra styles
isAxes = 0; % true if control is an axes
if isfield(r, 'Style')
 % translate panel to frame:
 if strcmp(r.Style, 'Panel')
  r.Style = 'Frame';
  r.ForegroundColor = r.BackgroundColor;
  r.Visible = 'off';
 elseif strcmp(r.Style, 'Axes')
  % style is an axes
  r = rmfield(r, {'Style', 'BackgroundColor'}); % remove the style and background color fields
  isAxes = 1;
 end
end

% check if control should become visible. Store the result in the visible flag.
if (isfield(r, 'Visible'))
 visible = r.Visible;
 r = rmfield(r, 'Visible');
else
 visible = 'on';
end

% translate 'r' to a cell array of form: {fieldname1, value1, fieldname2, value2, ...}:
values = struct2cell(r);
names = fieldnames(r);
uiargs = {};
for i=0:length(names)-1,
 uiargs{i*2+1} = names{i+1};
 uiargs{i*2+2} = values{i+1};
end

% create the control hidden:
if (isAxes)
 figure(d.hfig); % make d.hfig the active figure
 result = axes('Units', 'pixels', 'Visible', 'off', uiargs{:}); % add an axes
else
 result = uicontrol(d.hfig, 'Units', 'pixels', 'Visible', 'off', uiargs{:}); % add an uicontrol
end

if (isfield(r, 'Style'))
 if ((strcmp(r.Style, 'Frame')) & (isfield(r, 'String')))
  % create frame with caption: idea put a text object on the frame border

  % we can only find the width and height of the text (ie the Extent) after creation of the text object
  % so first create it invisible, ask the size, move the frame down to make room for the caption, set the caption position and
```

```
    % at last make it visible if the frame is visible.
    hcaption = uicontrol('Style', 'Text', 'String', r.String, 'BackgroundColor', get(result, 'BackgroundColor'),...
                                                    'ForegroundColor', get(result, 'ForegroundColor'), 'Visible', 'Off');
    textsize = get(hcaption, 'Extent');

    % store the caption the the user date field of the frame and move the frame down to make place for the caption:
    set(result, 'UserData', hcaption);
    p = get(result, 'Position');
    set(result, 'Position', [p(1), p(2)-textsize(4)/2, p(3), p(4)]);

    set(result, 'Visible', visible);
    p = setpos(d, hcaption, result, [d.Indent(1),-textsize(4)/2,textsize(3),textsize(4)]);
    set(hcaption, 'Position', p, 'Visible', visible);
  end
end

% show the new control if the visible flag was set:
set(result, 'Visible', visible);


function result = inRegion(xy, r)
% true if point xy is in box r. xy is a list [x,y] and r a box [left,top,width,height]
result = ((xy(1) >= r(1)) & (xy(2) >= r(2)) & (xy(1) < r(1) + r(3)) & (xy(2) < r(2) + r(4)));

function result = mouseClick(d)
% called when user presses mouse button somewhere in the figure window
xy = get(d.hfig, 'CurrentPoint'); % get position of last mouseclick
if (inRegion(xy, get(d.hInputimage, 'Position')))
    % click on the input image -> show it in a seperate window
    result = viewImg(TImage(d.img), 'input image', d);
elseif (inRegion(xy, get(d.hOutputimage, 'Position')))
    % click on the output image -> show it in a seperate window
    result = viewImg(d.outimg, 'output image', d);
else
  result = d;
end;

function result = viewImg(img, imgTitle, d)
% display an image IMG of type TImage in a seperate window centered on the screen
% and set the window's title to imgTitle.
rhimage(img, '');
result = d;

function result = sigmaEditEnterPressed(d)
% user pressed enter in the sigma value input box

sd = str2num(get(d.hSigmaEdit, 'String')); % get text in input box

if ((length(sd) == 0) | (sd <= 0) | (sd > 32))
  % error: user didn't type a valid number -> put current value of
  % SD back in edit box
  set(d.hSigmaEdit, 'String', num2str(d.sigma));
else
  % update value of sigma
  d.sigma = sd;

  % invalidate computated gradient
  d.Lx = '';
  d.Ly = '';
  d.winners = '';
end
result = d;

function result = sigmaLowEditEnterPressed(d)
% user pressed enter in the sigma low value input box

sd = str2num(get(d.hSigmaLowEdit, 'String')); % get text in input box

if ((length(sd) == 0) | (sd <= 0) | (sd > 32))
  % error: user didn't type a valid number -> put current value of
  % SD back in edit box
  set(d.hSigmaLowEdit, 'String', num2str(d.sigmalow));
else
  % update value of sigma
  d.sigmalow = sd;
```

```
  % invalidate computated gradient
  d.winners = '';
  d.Lx = '';
  d.Ly = '';
end
result = d;

function result = sigmaHighEditEnterPressed(d)
% user pressed enter in the sigma high value input box

sd = str2num(get(d.hSigmaHighEdit, 'String')); % get text in input box

if ((length(sd) == 0) | (sd <= 0) | (sd > 32))
  % error: user didn't type a valid number -> put current value of
  % SD back in edit box
  set(d.hSigmaHighEdit, 'String', num2str(d.sigmahigh));
else
  % update value of sigma
  d.sigmahigh = sd;

  % invalidate computated gradient
  d.Lx = '';
  d.Ly = '';
  d.winners = '';
end
result = d;

function result = alphaEditEnterPressed(d)
% user pressed enter in the alpha value input box

a = str2num(get(d.hAlphaEdit, 'String')); % get text in input box

if ((length(a) == 0) | (a <= 0))
  % error: user didn't type a valid number -> put current value of
  % alpha back in edit box
  set(d.hAlphaEdit, 'String', num2str(d.alpha));
else
  % update value of alpha
  d.alpha = a;
end
result = d;

function result = k1EditEnterPressed(d)
% user pressed enter in the k1 value input box

a = str2num(get(d.hK1Edit, 'String')); % get text in input box

if ((length(a) == 0) | (a <= 0))
  % error: user didn't type a valid number -> put current value of
  % k1 back in edit box
  set(d.hK1Edit, 'String', num2str(d.k1));
else
  % update value of k1
  d.k1 = a;
end
result = d;

function result = k2EditEnterPressed(d)
% user pressed enter in the k2 value input box

a = str2num(get(d.hK2Edit, 'String')); % get text in input box

if ((length(a) == 0) | (a <= 0))
  % error: user didn't type a valid number -> put current value of
  % k2 back in edit box
  set(d.hK2Edit, 'String', num2str(d.k2));
else
  % update value of k2
  d.k2 = a;
end
result = d;

function result = thresholdLowEditEnterPressed(d)
% executed when user presses enter in the threshold low edit box
```

```
thresholdLow = str2num(get(d.hThresholdLowEdit, 'String')); % get text in input box

if ((length(thresholdLow) == 0) | (thresholdLow < 0) | (thresholdLow > 255))
  % error: user didn't type a valid number -> put current value of
  % threshold back in edit box
  set(d.hThresholdLowEdit, 'String', num2str(d.thresholdLow));
else
  % update value
  d.thresholdLow = thresholdLow;
end
result = d;

function result = thresholdEditEnterPressed(d)
% executed when user presses enter in the threshold edit box
threshold = str2num(get(d.hThresholdEdit, 'String')); % get text in input box

if ((length(threshold) == 0) | (threshold < 0) | (threshold > 255))
  % error: user didn't type a valid number -> put current value of
  % threshold back in edit box
  set(d.hThresholdEdit, 'String', num2str(d.threshold));
else
  % update value
  d.threshold = threshold;
end
result = d;

function result = thinningChkClk(d)
% executed when user clicks the thinnen checkbox
d.thinnen = get(d.hThinningChk, 'Value'); % get checkbox state
result = d;

function result = brightnessChanged(d)
% executed when user changes the brightness
d.brightness = get(d.hBrightnessCombo, 'Value') - 1; % get combobox state
result = d;

function result = thresholdComboChanged(d)
% executed when user changes the threshold combobox
d.thresholdMethod = get(d.hThresholdCombo, 'Value') - 1; % get combobox state

% get width and height of figure (w,h) and the figures center (w2,h2)
dims = get(d.hfig, 'Position');
w = dims(3);
h = dims(4);
w2 = fix(w/2);
h2 = fix(h/2);

% show necessary threshold edit boxes
p = getpos(d.hThresholdEdit);
if d.thresholdMethod > 1
  % hysteresis thresholding: show TLOW edit box, set tooltip string of THIGH edit box
  set(d.hThresholdLowEdit, 'Visible', 'on');
  set(d.hThresholdEdit, 'Position', setpos(d, d.hThresholdEdit, {'rightof', d.hThresholdLowEdit, p(3), p(4)}));
  set(d.hThresholdEdit, 'TooltipString', d.ThresholdHighEditTooltipTxt);
else
  % no hysteresis thresholding: hide TLOW edit box, set tooltip string of the threshold edit box
  set(d.hThresholdLowEdit, 'Visible', 'off');
  set(d.hThresholdEdit, 'Position', setpos(d, d.hThresholdEdit, {'rightof', d.hThresholdCombo, p(3), p(4)}));
  set(d.hThresholdEdit, 'TooltipString', d.ThresholdEditTooltipTxt);
end
if d.thresholdMethod > 0
  % apply (hysteresis) thresholding: hide the brightness combobox and show the (HIGH) threshold edit box
  set(d.hBrightnessCombo, 'Visible', 'off');
  set(d.hThresholdEdit, 'Visible', 'on');
else
  % no thresholding: hide the threshold edit box and show the brightness combobox
  set(d.hThresholdEdit, 'Visible', 'off');
  set(d.hBrightnessCombo, 'Visible', 'on');
end
result = d;
function result = closeFigure(d)
% called if user presses the close button

freePImage(d.img); % free the input image
delete(get(0,'CurrentFigure')) % close the figure
```

```
result = []; % mark structure d as invalid

function result = SIAlphaEditEnterPressed(d)
% called if enter pressed in surround inhibition alpha editbox
siAlpha = str2num(get(d.hSIAlphaEdit, 'String')); % get text in input box

if ((length(siAlpha) == 0) | (siAlpha <= 0))
 % error: user didn't type a valid number -> put current value of
 % threshold back in edit box
 set(d.hSIAlphaEdit, 'String', num2str(d.siAlpha));
else
 % update value
 d.siAlpha = siAlpha;
end
result = d;

function result = InputImgComboClk(d)
% called if the input image combobox is clicked

i = get(d.hInputImgCombo, 'Value') - 1; % get combobox state

% free old input image
freePImage(d.img);

% load input image
switch (i)
 case 0
  d.filename = 'contourdemo/lena.jpg';
 case 1
  d.filename = 'contourdemo/gnu_2.pgm';
 case 2
  d.filename = 'contourdemo/blocks.bmp';
end
d.img = PImage(rhloadimage(d.filename));

% show input image and hide its axes
rhimage(TImage(d.img), d.hInputimage);

% invalidate computated gradient
d.Lx = '';
d.Ly = '';
d.winners = '';

result = d;

function result = InputImgBtnClk(d)
% called if the input image button is clicked

% open a file browser
[filename, path] = uigetfile('*.jpg;*.pgm', 'Select an input image...');

if (filename ~= 0)
 % ok clicked -> load the image file and show it
 d.filename = [path  filename];
 freePImage(d.img);
 d.img = PImage(rhloadimage(d.filename));
 rhimage(TImage(d.img), d.hInputimage);

 % invalidate computated gradient
 d.Lx = '';
 d.Ly = '';
 d.winners = '';
end

result = d;

function result = SaveBtnClk(d)
% called if the save button is clicked


% open a file browser
[filename, path] = uiputfile('output.jpg', 'Save output image as...');

if (filename ~= 0)
 % ok clicked -> save the image file
```

```
 saveimg(d.outimg, [path filename]);
end

result = d;

function result = GradientCalcMethodComboClk(d)
% called if gradient calculation method combobox selection changes

% invalidate computated gradient
d.gradient = '';
d.Lx = '';
d.Ly = '';
d.winners = '';

% set gradient computation method variable and show correct edit boxes
if (get(d.hGradientCalcMethodCombo, 'Value') == 1)
 d.gradientcomputation = 'single-scale';
 set(d.hSigmaEdit, 'Visible', 'on');
 set(d.hSigmaLowEdit, 'Visible', 'off');
 set(d.hSigmaHighEdit, 'Visible', 'off');
else
 d.gradientcomputation = 'multi-scale';
 set(d.hSigmaEdit, 'Visible', 'off');
 set(d.hSigmaLowEdit, 'Visible', 'on');
 set(d.hSigmaHighEdit, 'Visible', 'on');
end

result = d;

function result = HelpBtnClk(d)
% opens the file ./contourdemo/index.html in the default browser

errcode = web([pwd, '/contourdemo/index.html']);

if (errcode ~= 0)
 fprintf('Error while opening help file: Can not start the web browser.\n');
end

result = d;
```

---

gaussian.m

```
function result = gaussian(img, sigma)
% GAUSSIAN Gaussian smoothing of an image
%  GAUSSIAN(IMG, SIGMA) applies a Gaussian filter with standard
%    diviation SIGMA to image img. IMG is of type PImage,
%    the function result of type TImage.
%
% see also: PIMAGE, TIMAGE, CONVOLVE

filter = gaussianConvolutionKernel2D(sigma); % construct the gaussian kernel. This is a
                                             % matrix.
result = convolve(img, filter);  % convolve the image with the gaussian filter.
ncolors = size(result, 3); % get amount of color components (1 =
                           % intensity image, 3 = color RGB image)
% clip the resulting blurred image to range [0, 1]
for c = 1:ncolors,
 result.img(:,:,c) = min(max(result.img(:,:,c),0.0),1.0);
end
```

---

gaussian2D.m

```
function result = gaussian2D(x, y, sigma)
% GAUSSIAN2D two dimensional Gaussian function
%  GAUSSIAN2D(X, Y, SIGMA) returns the Gaussian function with standard
%    deviation SIGMA at each location (X(i), Y(i)). X and Y and function
%    result are matrices of any, but equal dimensions.
%    So for example to get the evaluate the Gaussian function with
%    a standard deviation of 2 at points (1,2) and (3,4) one uses the call
```

```
%    gaussian2D([1 3], [2 4], 2). This would yield [0.0101 0.0058].

result = exp(-(x.*x + y.*y) / (2 * sigma*sigma)) / (2 * pi * sigma*sigma);
```

---

gaussianConvolutionKernel2D.m

```
function result = gaussianConvolutionKernel2D(sigma)
% GAUSSIANCONVOLUTIONFILTER2D 2D Gaussian convolution kernel
%   GAUSSIANCONVOLUTIONFILTER2D(SIGMA) creates a gaussian kernel,
%     a matrix, with standarddeviation SIGMA.
%
% See also: CONVOLVE

n = 3 * ceil(sigma) - 1; % filter size is n to left and n to right from filter center
[xcoords, ycoords] = meshgrid(-n:n); % create a grid with x and y coordinates in range -n:n
result = gaussian2D(xcoords, ycoords, sigma); % calculate the gaussian at each gridpoint
result = normalize(result); % normalizes the matrix with the L1-norm
```

---

gradientOfGaussian.m

```
function [Lx, Ly] = gradientOfGaussian(img, sigma, normalizekernel)
% GRADIENTOFGAUSSIAN Gaussian smooths an image and returns the gradient
%   GRADIENTOFGAUSSIAN(IMG, SIGMA, NORMALIZEKERNEL) calculates the gradient
%     [dIg/dx,dIg/dy] where Ig is the image IMG smoothed by a Gaussian function
%     with standard deviation SIGMA. IMG must be of type PImage. The two
%     function results are both of type TImage. If NORMALIZEKERNEL
%     is nonzero then the kernel is normalized using the L1-norm.
%
% see also: CONVOLVE, PIMAGE, TIMAGE

lx = convolve(img, derivativeOf2DGaussianKernel(sigma, 1, normalizekernel)); % convolve img with dG/dx
ly = convolve(img, derivativeOf2DGaussianKernel(sigma, 2, normalizekernel)); % convolve img with dG/dy

% results are only accurate to 10 digits or so, so round gradient to this precision
lx.img = round(lx.img * 1e10) / 1e10;
ly.img = round(ly.img * 1e10) / 1e10;

% if the input image is a color image, then there are 3 gradients per pixel
% (one per colorcomponent R,G,B). We wish to return only a single gradient
% per pixel. We return per pixel the gradient with maximal gradient magnitude
% among the three:
ncolors = size(lx.img, 3);
if (ncolors == 3)
  % we have a color image, as there are 3 color components

  % get the gradient magnitude per color component
  M = getGradientMagnitude(lx, ly);

  % Note: The code below makes use of the fact that
  % (i-1) * A + (2-i) * B equals per pixel (x,y) A(x,y) if i(x,y)=2 and B
  % if i(x,y)=1, assuming matrix i contains solely ones and twos.

  % first compare the gradient magnitude of the first and second colorcomponent
  % per pixel. Use per pixel the maximum of the two, this is (Lx, Ly). The
  % gradient magnitude of (Lx,Ly) is M2. indexes(x,y) = 1 if M(:,:,1) is larger
  % than M(:,:,2) and 2 otherwise.
  [M2, indexes] = maximg(TImage(M.img(:,:,1)), TImage(M.img(:,:,2)));
  Lx = TImage((indexes-1) .* lx.img(:,:,2) + (2 - indexes) .* lx.img(:,:,1));
  Ly = TImage((indexes-1) .* ly.img(:,:,2) + (2 - indexes) .* ly.img(:,:,1));

  % now compare M2 with the third colorcomponent.
  % Use per pixel the maximum of the two, this is (Lx, Ly). The
  % gradient magnitude of (Lx,Ly) is M2. indexes(x,y) = 1 if M2 is larger
  % than M(:,:,3) and 2 otherwise.
  [M2, indexes] = maximg(M2, TImage(M.img(:,:,3)));
  Lx = TImage((indexes-1) .* lx.img(:,:,3) + (2 - indexes) .* Lx.img);
  Ly = TImage((indexes-1) .* ly.img(:,:,3) + (2 - indexes) .* Ly.img);
else
  % the input image is an intensity image -> just return the calculated gradient
```

```
 Lx = lx;
 Ly = ly;
end
```

---

derivativeOf2DGaussianKernel.m

```
function result = derivativeOf2DGaussianKernel(sigma, dimension, normalizekernel)
% DERIVATIVEOF2DGAUSSIANFILTER constructs a dG filter
%  DERIVATIVEOF2DGAUSSIANFILTER(SIGMA, DIMENSION, NORMALIZEKERNEL) constructs the
%    filter dG/dx if DIMENSION is 1 and dG/dy if DIMENSION is 2. G is
%    the gaussian filter with standard deviation SIGMA. The result
%    is a 2D matrix. If NORMALIZEKERNEL is nonzero then the kernel is normalized
%    using the L1-norm.
%    Note: If the kernel is normalized, it gives, when applied
%    to a step edge, a constant value of 0.5 (independant of sigma).
%
% See also: CONVOLVE

% The function will be initialized in a window of size 2n+1 x 2n+1:
n = 3 * ceil(sigma);

% Create a grid with all sample point coordinates. Each pair (xcoord(i), ycoord(i)) is
% a sample point.
[xcoords, ycoords] = meshgrid(-n:n);

% Sample the dG/dx or dG/dy function at all sample points:
if (dimension == 1)
 result = -xcoords .* exp(-((xcoords.^2 + ycoords.^2)/(2*sigma^2)))/(sqrt(2*pi)*sigma^3);
elseif (dimension == 2)
 result = -ycoords .* exp(-((xcoords.^2 + ycoords.^2)/(2*sigma^2)))/(sqrt(2*pi)*sigma^3);
end

if (normalizekernel)
 result = normalize(result); % Normalize the filter with the L1-norm
end
```

---

multiscale.m

```
function [Lx, Ly, Winners] = multiscale(img, sigmas)
% MULTISCALE gradient computation using multiple scales
%  [dIMG/dx, dIMG/dy, WINNERS] = MULTISCALE(IMG, SIGMAS)
%    Gradient computation using multiple scales.
%    SIGMAS is the list of values of sigma to use. The best results
%    are achieved if a list of the form mkLogList(1,h,sqrt(2)) is used
%    where h is a power of 2.
%    The gradient images created using the specified values of sigma
%    are combined into one gradient image, which is returned as
%    (dIMG/dx, dIMG/dy). IMG must be of type PImage, the function
%    results dIMG/dx and dIMG/dy are of type TImage.
%    WINNERS is a matrix of the same dimensions as the input
%    image, containing per pixel a value i if SIGMAS(i) was used in the final result.
%
%  Example: Apply multiscaling to the given image file and display the result using
%         sigmas [sqrt(2)^0,..,sqrt(2)^n,8]:
%
%    [lx, ly] = multiscale(PImage(rhloadimage('imagefile')), mkLogList(1,8,sqrt(2)));
%    rhimage(getGradientMagnitude(lx, ly));
%
% see also: DGOP, GRADIENTOFGAUSSIAN, MKLOGLIST, PIMAGE, TIMAGE

% Idea: calculate the gradient of the image per sigma. Normalize
% the results in such a way that a large sigma gives the largest
% response (ie gradient magnitude) on a wide edge and via versa.
% Using an image with increasingly wide edges as
% created by MKMULTISCALEIMG, one can see that the nomalization
% factor should be a division by sigma^0.5.
% Per sigma the gradient magnitude and the gradient image with thinned edges
% are calculated (ie M and Mthinned).
% In each step in the for-loop, we try to get from the results
```

```
% produced by using only sigmas(1)..sigmas(i-1) to the result when using
% sigmas(1)..sigmas(i). We maintain the maximum gradient per pixel
% among the gradient magnitude image produced thus far in [Lx,Ly],
% and |[Lx,Ly]| in HighestM. In the variable 'result' the combined gradient
% magnitude image in which edges are thinned is maintained.
% The sigma which is used at each pixel in [Lx, Ly] is stored in matrix 'Winners2'.
n = length(sigmas);
for i = 1:n,
 % calculate the gradient (gx, gy) using sigmas(i) and normalize
 [gx, gy] = gradientOfGaussian(img, sigmas(i), 0);
 gx.img = gx.img / (sigmas(i)^0.5);
 gy.img = gy.img / (sigmas(i)^0.5);

 % store the gradient magnitude image in M and the same with
 % thinned edges in Mthinned.
 M = getGradientMagnitude(gx, gy);
 Mthinned = thinnenEdgesSimple(gx, gy);

 if (i == 1)
  % lowest sigma used:
  % As there's no smaller sigma used thus far, HighestM equals M and
  % 'result' equals Mthinned. (HighestM is the pixelwise maximum of
  % the gradient magnitudes images calculated thus far). Lx and
  % Ly are of course equal to gx and gy, Winners2 equals sigmas(1)
  % and Winners equals 1.
  HighestM = M;
  result = Mthinned;
  Lx = gx;
  Ly = gy;
  Winners2 = ones(size(result.img)) .* sigmas(i);
  Winners = ones(size(result.img));
 else
  % i > 1:
  % HighestM is the pixelwise maximum of the gradient magnitudes
  % images calculated thus far, so it equals the pixelwise maximum of the
  % previous HighestM and M.
  HighestM = maximg(HighestM, M);

  % 'result' can be extended by taking the pixelwise maximum
  % of the previous result and Mthinned.
  [result, indexes] = maximg(result, Mthinned);

  % We can now update Winners, Winners2, Lx and Ly by
  % keeping the old values if 'result' is unchanged at that pixel
  % else they are changed respectively to sigmas(i) and the value of
  % gx and gy at that pixel. We can check this with the indexes
  % variable which is one if 'result' is unchanged at a pixel and
  % two otherwise.
  % Note: The code below makes use of the fact that
  % (i-1) * A + (2-i) * B equals per pixel (x,y) A(x,y) if i(x,y)=2 and B
  % if i(x,y)=1, assuming matrix i contains solely ones and twos.
  Winners = (indexes-1) .* i + (2 - indexes) .* Winners;
  Winners2 = (indexes-1) .* sigmas(i) + (2 - indexes) .* Winners2;
  Lx.img = (indexes-1) .* gx.img + (2 - indexes) .* Lx.img;
  Ly.img = (indexes-1) .* gy.img + (2 - indexes) .* Ly.img;
 end;

 % clear the temporary variables
 clear Mthinned;
 clear M;
 clear gx;
 clear gy;
 clear indexes;
end

% A gradient magnitude in 'result' is set to zero, if it
% isn't maximal among all gradient magnitude images.
% As HighestM keeps track of the pixelwise maximum,
% we can check per pixel if the magnitude is maximal by
% the array operand (result.img >= HighestM.img-1e-10).
% The -1e-10 is needed to compensate for computational noise.
% This gives per pixel a one (true) or zero (false). So by
% multiplying this to the result itself, we get what we want.
result.img = (result.img >= HighestM.img-1e-10) .* result.img;
clear HighestM; % we don't need this variable any more
```

```
% Now thinnen the edges from gradient image [Lx, Ly]. Let's call
% the resulting gradient magnitude image TL. We set a pixel's value
% to zero in both Lx and Ly if the pixel's value is zero in 'result' or in TL.
% At last: multiplying Lx and Ly by Winners.^0.5 makes sure that edges with
% different width, but at which the same change in intensity occurs get the
% same gradient magnitude.
M = (getfield(thinnenEdgesSimple(Lx, Ly), 'img') > 1e-10) & (result.img > 1e-10);
Lx.img = M .* Lx.img .* (Winners2.^0.5);
Ly.img = M .* Ly.img .* (Winners2.^0.5);
```

---

mkmultiscaleimg.m

```
function result=mkmultiscaleimg(w, h, sigmalow, sigmahigh, k)
% MKMULTISCALEIMG creates a test image for multiscaling
%  MKMULTISCALEIMG(W, H, SIGMALOW, SIGMAHIGH, K) creates an image
%    (ie a matrix) of size H x W. This image is devided in two.
%    The left half is made black (ie zero), the right half white
%    (ie one). Groups of rows in this image are now convolved with
%    a Gaussian filter in the following way. The values of sigma
%    used are SIGMALOW, K SIGMALOW, K^2 SIGMALOW, ..., SIGMAHIGH.
%    Suppose there are N elements in this list of sigmas then the
%    first H/N lines are convolved with a Gaussian filter with
%    standard deviation SIGMALOW, the next H/N with K SIGMALOW, etc.
%    The resulting intensity image, a matrix of size H x W of type TImage, is
%    returned. A much used value of K is sqrt(2). SIGMALOW and K must be positive.
%    SIGMAHIGH must be at least SIGMALOW.
%
% see also: GAUSSIAN, RHIMAGE, TIMAGE

% compute the sigmas to use:
sigmas = mkLogList(sigmalow, sigmahigh, k);

% now create the image consisting of a black and white half. This
% can be done by creating a matrix of zeros with the zeros function
% and one consisting of ones with the ones function. Then these
% two matrices can be joined. The result is stored in IMG.
img = [zeros(h, fix(w / 2)), ones(h, round(w / 2))];

% set N to the amount of sigmas with which the image IMG is
% convolved. The image should be grouped in blocks of rows of equal
% size. This size, h/n, is called BLOCKSIZE. Now block i is convolved
% with a Gaussian with standard deviation SIGMAS(i). The last block
% is dealt with seperately as this block is possible larger than
% the other ones as h/n might have a remainder, which is given to
% the last block.
n = length(sigmas);
blocksize = fix(h / n);
result = TImage(zeros(h,w)); % create a TImage of size H x W for the result
if (blocksize ~= 0)
 for i = 0:n-2,
  rows = PImage(img(i*blocksize+1:(i+1)*blocksize, :)); % rows to convolve

                % with sigmas(i+1)
  result.img(i*blocksize+1:(i+1)*blocksize, :) = getfield(gaussian(rows, sigmas(i+1)), ...
                                                  'img'); % convolve the rows
  freePImage(rows); % free memory used by PImage ROWS
 end
end

rows = PImage(img((n-1)*blocksize+1:h, :)); % rows to convolve with sigmas(n)
result.img((n-1)*blocksize+1:h, :) = getfield(gaussian(rows, sigmas(n)), ...
                                                  'img'); % convolve the rows
freePImage(rows); % free memory used by PImage ROWS
```

```
function result = thinnenEdgesLinear(lx, ly)
% THINNENEDGESLINEAR non-maximum suppression using lineair interpolation scheme
%  THINNENEDGESLINEAR(LX, LY) makes the edges in the gradient
%    magnitude image of gradient [LX, LY] thinner by means of
%    non-maximum suppression. (LX,LY) are the,
%    gradients of the input image at each point.
%    LX, LY and function result must be intensity images of type TImage.
%
% See also: TImage

% we assume that because of computational errors, the gradient (lx,ly) is only accurate
% up to 10 digits, so we round it to 10 digit precision before continueing.
% Without this several problems occur, for example if the gradient of a vertical white bar
% is calculated, ly should be zero. In fact, because numbers have limited precision,
% it ranges between [-1e-14,1e-14]. So for example a test ly == 0 would not make sense.
% See also the functions RHSIGN and RHFIX.
lx.img = round(lx.img * 1e10) / 1e10;
ly.img = round(ly.img * 1e10) / 1e10;

% calculate the gradient magnitude (M = sqrt(lx^2 + ly^2)):
M = sqrt(lx.img .^ 2 + ly.img .^ 2);

% thinnen the edges
result = thinnen(M, lx.img, ly.img);

% translate the result to a TImage
result = TImage(result);

function result = thinnen(M, lx, ly)
% performs the real edge thinning

% suppose we want to make the discrete input image continues. In order to do this we look at the pixels as being
% squares with width and height 1. The pixels only take their color values at their centers.
% We define the pixel border is the square through a pixels neighbours (see ascii art below).
%
% Per pixel (x0,y0) we need to determine which pixelborders are intersected by the line L(x0,y0) through
% points (x0,y0) and (x0,y0) + (lx,ly)(x0,y0). if there's no intersection, ie both lx(x0,y0) and ly(x0,y0) are zero,
% then the magnitude is 0, so there can't be an edge at pixel (x0,y0)(x0,y0) in that case.
% otherwise there are 2 intersectionpoints at which the (interpolated) gradient magnitudes are A(x0,y0) and B(x0,y0).
% The values A(x0,y0) and B(x0,y0) are found by linear interpolation between the gradient magnitudes of the two pixels next
their
% locations on the pixel border.
% If M(x0,y0)(x0,y0) is larger than A(x0,y0) and if it is at least B(x0,y0) then we have a local maximum at (x0,y0) and
% it is an edge pixel and its gradient magnitude value should remain unchanged else it is set to zero.
%
%
% --|--------|--------|---------|--   * = pixel centers
%   |        |        |         |     squares are pixels
%   |   *,,,,,,,,*,,,,,,A,,*    |     ,,, = pixel border of center pixel
%   |   ,    |  /|\   |    ,    |     A, B: the intersectionpoints of the vector
%                                        (lx,ly) with the pixel border
% --|---,----|---|----|----,----|--
%   |   ,    |lx |  ly| \  ,    |
%   |   *    |    * ------  *    |
%   |   ,    | (x0,y0)| /  ,    |
% --|---,----|--------|----,----|--
%   |   ,    |        |    ,    |
%   |   *,B,,,,,,,,,,,,,,,*    |
%   |        |        |         |
% --|--------|--------|---------|--

% The idea is to find the intersection points of L(x0,y0) with the line extensions of the border of (x,y) first.
% There are 2 points in the direction of the gradient, namely one intersection with a horizontal border line, Ah(x,y) and
% one with a vertical Av(x0,y0). In the opposite direction we get Bh(x,y) and Bv(x,y).
% Now check which intersection point Ah(x0,y0) or Av(x0,y0) lays on the pixel border. Its gradient magnitude is A(x0,y0).
% Simulary we find B(x0,y0). This can be done using linear algebra.

warning off; % suppress devision by zero warnings

h = size(M,1); % height of image
w = size(M,2); % width of image

% store all pixel coordinates. each pair of x0 and y0 is an coordinate
```

```
x0 = transpose(reshape(mod([0:w*h-1], w), w, h)+2); % (2:w+1) h times
y0 = transpose(reshape(fix([0:w*h-1] / w), w, h)+2); % 2 w times .. h+1 w times

% calculate the direction of each pixel's gradient /=1 \=-1
dir = 1 - 2 * xor(lx < 0, ly < 0);

% determine ly / lx and lx / ly
lxly = lx ./ ly;
lylx = ly ./ lx;

% add a border of zeros to M to make calculations easier the result is Mold:
Mold(h+2,w+2) = 0;
Mold(2:h+1,2:w+1) = M(1:h,1:w);

% determine A:

% interpolate Ah = p * m1 + (1-p) * m2
% p = horz distance between intersection of the gradient vector with
% the line extension of the top pixel border.
% if the outcome of p is NaN its mapped to 2 as in this case there's
% no intersection. Ah then isn't used in the calculation of A
% (see A = ...)
% m1 and m2 are the gradient magnitudes of the pixels left and right
% of the intersection point.
p = mkvalid(lxly .* (y0 - 1) + x0 - lxly .* y0 - x0, 2); % find p. Map non-finite results to 2.
j = sub2ind(size(Mold), y0-1, (x0)); % find the coordinates of the pixel above (x0,y0). Its index is j
m1 = reshape(Mold(j), h, w); % the gradient magnitude of pixel j is m1.
j = sub2ind(size(Mold), (y0-1), (x0+rhsign(p))); % get the coordinates of the other pixel next to the intersection on the pixel
border
m2 = reshape(Mold(j), h, w); % its gradient magnitude is m2
p = abs(p); % interpolate the value of Ah using formula Ah = |p| * m1 + (1-|p|) * m2
Ah = mkvalid(p .* m1 + (1 - p) .* m2, 0); % set all nonfinite values in Ah to 0

% interpolate Av = q * m1 + (1-q) * m2
% q = vertical distance between intersection of the gradient vector with
% the line extension of the left (if dir = /) or right (if dir = \) pixelborder
% if the outcome of q is NaN its mapped to 2 as in this case there's
% no intersection. Av then isn't used in the calculation of A
% (see A = ...)
% m1 and m2 are the gradient magnitudes of the pixels above and beneath
% the intersection point.
q = mkvalid(lylx .* (x0 + dir) + y0 - lylx .* x0 - y0, 2); % find q. Map non-finite results to 2.
j = sub2ind(size(Mold), (y0), (x0 + dir)); % find the coordinates of the pixel left/right of (x0,y0) depending on the intersection
border
m1 = reshape(Mold(j), h, w); % the gradient magnitude of pixel j is m1.
j = sub2ind(size(Mold),(y0 + rhsign(q)), (x0 + dir)); % get the coordinates of other pixel next to the intersection on the pixel
border
m2 = reshape(Mold(j), h, w); % its gradient magnitude is m2
q = abs(q); % interpolate the value of Av using formula Ah = |q| * m1 + (1-|q|) * m2
Av = mkvalid(q .* m1 + (1 - q) .* m2, 0); % set all nonfinite values in Av to 0

% check which border, horz (if p<=1) or vert (if q<=1), is intersected. (only one of the
% 2 tests below will give a nonzero result per pixel).
A = (p <= 1) .* Ah + (q < 1) .* Av;
clear Ah;
clear Av;

% determine B.
% This is simular to the determination of A, but now we look in the opposite direction of the gradient
% at each pixel:

% Find Bh:
p = mkvalid(lxly .* (y0 + 1) + x0 - lxly .* y0 - x0, 2);
j = sub2ind(size(Mold), (y0+1), (x0));
m1 = reshape(Mold(j), h, w);
j = sub2ind(size(Mold), (y0+1), (x0+rhsign(p)));
m2 = reshape(Mold(j), h, w);
p = abs(p);
Bh = mkvalid(p .* m1 + (1 - p) .* m2, 0);

% Find Bv:
dir = -dir;
q = mkvalid(lylx .* (x0 + dir) + y0 - lylx .* x0 - y0, 2);
j = sub2ind(size(Mold), (y0), (x0 + dir));
m1 = reshape(Mold(j), h, w);
```

```
j = sub2ind(size(Mold),(y0 + rhsign(q)), (x0 + dir));
m2 = reshape(Mold(j), h, w);
q = abs(q);
Bv = mkvalid(q .* m1 + (1 - q) .* m2, 0);

% check which border (horz or vert) is intersected (only one of the
% 2 tests below will give a nonzero result per pixel).
B = (p <= 1) .* Bh + (q < 1) .* Bv;
clear Bh;
clear Bv;

% there's a local maximum if the magnitude at a pixel (x,y) is larger than A(x,y)
% and larger of equal than B(x,y) in the direction of the gradient.
% Note: 1E-10 is added to counteract computational errors
result = ((M > A+1e-10) & (M > B-1e-10)) .* M;


warning on; % turn warning messages back on
```

---

thinnenEdgesSimple.m

```
function result = thinnenEdgesSimple(lx, ly)
% THINNENEDGESSIMPLE edge thinning using non-maximum suppression
%   THINNENEDGESSIMPLE(LX, LY) makes the edges in the gradient
%     magnitude image of gradient (LX, LY) thinner by means of
%     non-maximum suppression. non-maximum suppression is performed by
%     checking the neighbouring pixels' magnitude in the direction
%     of the gradient. LX, LY and function result are TImages
%     A more accurate form of edge thinning can be done with
%     THINNENEDGESLINEAR.
%
% see also: GRADIENTOFGAUSSIAN, THINNENEDGESLINEAR, TIMAGE

% we assume that because of computational errors, the gradient (lx,ly) is only accurate
% up to 10 digits, so we round it to 10 digit precision before continueing.
% Without this several problems occur, for example if the gradient of a vertical white bar
% is calculated, ly should be zero. In fact, because numbers have limited precision,
% it ranges between [-1e-14,1e-14]. So for example a test ly == 0 would not make sense.
% See also the functions RHSIGN and RHFIX.
lx.img = round(lx.img * 1e10) / 1e10;
ly.img = round(ly.img * 1e10) / 1e10;

% calculate the gradient magnitude (M = sqrt(lx^2 + ly^2)):
M = sqrt(lx.img .^ 2 + ly.img .^ 2);

% thinnen the edges:
result = thinnen(M, lx.img, ly.img);

% translate the result to a TImage
result = TImage(result);

function result = thinnen(M, lx, ly)
% thinnens the edges

warning off; % suppress division by zero warnings

h = size(M,1); % height of image
w = size(M,2); % width of image

% store all pixel coordinates. each pair of x0 and y0 is an image coordinate
x0 = transpose(reshape(mod([0:w*h-1], w), w, h)+2); % (2:w+1) h times
y0 = transpose(reshape(fix([0:w*h-1] / w), w, h)+2); % 2 w times .. h+1 w times

% add a border of zeros to the matrix M to make calculations easier, call the result Mb
Mb(h+2,w+2) = 0;
Mb(2:h+1,2:w+1) = M(1:h,1:w);

% Retreive magnitudes A, B of the neighbouring pixels in both directions
% of the gradient at each pixel. The neighbouring pixels are those
% crossed by the gradient vector. So first calculate the angle
% between the vector (lx,ly) and the x-axes per pixel, alpha.
% If the neighbour in the gradient direction is right to this pixel,
```

```
% then dx is 1, if its on the left -1, above/beneath 0. Simulary
% we define dy as 1 if its below the pixel, -1 if its above, 0 if
% it's left or right of the pixel. So the neighbour in the
% direction of the gradient is (x0+dx,y0+dy) and the one in the
% opposite direction (x0-dx,y0-dy). The correct values of dx and dy
% can be found by checking alpha and in case of dy the sign of ly as well.
% Note: if M=0 then we do not care what the value of dx and dy are;
% there can't be a local maximum at the pixel.
alpha = acos(lx ./ M);
dx = (alpha < (3/8)*pi) - (alpha > (5/8)*pi);
dy = ((alpha > (1/8)*pi) & (alpha < (7/8)*pi)) .* mkvalid(sign(ly),0);
A = Mb(sub2ind(size(Mb), y0 + dy, x0 + dx));
B = Mb(sub2ind(size(Mb), y0 - dy, x0 - dx));

% there's a local maximum if the magnitude at a pixel (x,y) is larger than A(x,y)
% and larger of equal than B(x,y) in the direction of the gradient. In that case we
% set result(x,y) to M(x,y) else it is set to zero:
% Note: 1E-10 is added to counteract computational errors
result = ((M > A+1e-10) & (M > B-1e-10)) .* M;

warning on; % display warning messages again
```

---

canny.m

```
function result = canny(img, sigma, tlow, thigh)
% CANNY canny edge detector
%   CANNY(IMG, SIGMA, TLOW, THIGH) executes the Canny edge
%     detector on IMG. First the image is convolved with a dG filter
%     with standard deviation SIGMA. Then the lines are thinned and at
%     last the result is thresholded using hysteresis thresholding with
%     TLOW and THIGH. TLOW and THIGH must have range [0, 1].
%     IMG must be of type PImage. The result is a TImage.
%
% see also: TIMAGE, PIMAGE, DGOP

% check if tlow and thigh have correct ranges
if ((tlow < 0) | (tlow > 1) | (thigh < 0) | (thigh > 1))
  error('tlow and thigh must have range [0, 1]');
end

% use the dGop function to perform the canny edge detection steps
result = dGop(img, sigma, 1, tlow, thigh, 0);
```

---

surroundInhibition.m

```
function result = surroundInhibition(M, alpha, k1, k2, sigmas, winners)
% SURROUNDINHIBITION suppresses edges close to others in a gradient magnitude image
%   C = SURROUNDINHIBITION(M, ALPHA, K1, K2, SIGMAS, WINNERS)
%     At each pixel (x,y) weight gradient magnitude image M by convolving it
%     with kernel HDoG_SIGMAS(i)_K1_K2 if in the construction of M, SIGMAS(i) was
%     used at this pixel (ie WINNERS(x,y) == i). Let us call the resulting matrix T.
%     HDoG_sigma_k1_k2 equals the kernel constructed by function
%     DoGConvolutionKernel2D(sigma, k1, k2), but negative values are mapped to zero
%     and the kernel is normalized.
%     Image C, the function result, is defined as H(M - ALPHA T), where ALPHA is a
%     weighting factor (0 <= ALPHA <= 1). Note that if no other edges
%     are present near pixel (x,y), the gradient magnitude of (x,y)
%     equals M(x,y). If a lot of edges are present, ALPHA*T is
%     large and so C(x,y) will become zero, so the operation
%     suppresses edges close to others in a gradient magnitude image.
%     M should be a PImage and C is a TImage. K1 should be larger than K2 otherwise
%     the result C equals TImage(M).
%
%   Example: Create a gradient magnitude image by applying
%   multiscaling to an image file. Then apply surround inhibition
%   and show the result:
%     img = PImage(rhloadimage('filename'));
%     sigmas = mkLogList(1, 8, sqrt(2));
%     [Lx, Ly, winners] = multiscale(img, sigmas);
```

```
%    M = PImage(getGradientMagnitude(Lx, Ly));
%    rhimage(surroundInhibition(M, 2, 4, 1, sigmas, winners));
%    freePImage(M);
%
% See also: DOGCONVOLUTIONKERNEL2D, TIMAGE, PIMAGE, MULTISCALE, DGOP

% First we initialize a matrix T with the same dimensions as WINNERS to zero.
% For each sigma used (ie for all sigmas(i)), create a DoG convolution kernel, DoG.
% Now map all negative values to zero and normalize the result.
% This gives normalized convolution kernel W.
% Now apply W to gradient magnitude image M, the result is an image C.
% Now multiply C by matrix (Winners == i) and add the result to T. As the matrix
% (Winners == i) is one at all pixels where i is the winner and zero at the other
% locations and as each pixel has only a single winner, after the FOR-loop, t(x,y)
% contains the convolution result of the DoG filter that uses the winning sigma at
% that pixel.
t = zeros(size(winners));
for i=1:length(sigmas),
  DoG = DoGConvolutionKernel2D(sigmas(i), k1, k2);
  w = normalize(DoG .* (DoG > 0));
  c = convolve(M, w);
  t = t + (c.img .* (winners == i));
end
clear w;
clear c;

% Weight T with ALPHA and substract it from M. Map all negative
% values to zero. This is C:
C = getfield(TImage(M), 'img') - alpha * t;
C = (C .* (C > 0));
clear t;
clear M;

% Translate C to a TImage. This is the function result:
result = TImage(C);
```

DoGConvolutionKernel2D.m

```
function result = DoGConvolutionKernel2D(sigma, k1, k2)
% DOGCONVOLUTIONKERNEL2D constructs a DoG convolution kernel
%   DOGCONVOLUTIONKERNEL2D(SIGMA, K1, K2) constructs a convolution kernel, a matrix, by
%   sampling the function DoG(x, y) = G_(K1*SIGMA)(x,y) - G_(K2*SIGMA)(x,y)
%   where G_sd is the Gaussian function with standard deviation 'sd'.
%   The convolution kernel is NOT normalized.
%
% See also: GAUSSIAN, CONVOLVE

% The filter is constructed by sampling the difference of Gaussian
% function mentioned above at every integer point. As
% the function is almost zero at SIGMA*(3*K1 + K2) from the origin, sampling
% at integer points at at most this distance from the origin suffices.

n = ceil(sigma)*(3*k1 + k2) - 1; % filter size is n to left and n to right from filter center
[x, y] = meshgrid(-n:n); % create a matrix (x,y) with the coordinates of the sample points

% apply the DoG function to each sample point
result = Gaussian2D(x, y, k1*sigma) - Gaussian2D(x, y, k2*sigma);
```

---

dGOp.m

```
function [result, winners] = dGop(img, sigmas, thinnen, thresholdLow, thresholdHigh, ...
                                  brightness, alpha, k1, k2)
% DGOP performs contour detection using the dG filter
%   [RESULT, WINNERS] = dGop(IMG, SIGMAS, THINNEN, THRESHOLDLOW, THRESHOLDHIGH,
%   BRIGHTNESS [,ALPHA, K1, K2])
%    This function performs all steps needed to do contour or edge
%    detection. It relies on most functions found in this
%    library. The many parameters are best understood by playing
%    with the supplied GUI, CONTOURDEMO and by reading its helpfile
```

```
%    (./contourdemo/index.html). Contour detection steps:
%    1) Edge enhancement: Transform IMG with the derivative of a
%       gaussian using given sigmas. If the list SIGMAS contains more
%       than one element then multiscaling is applied. With
%       multiscaling edgethinning is always applied and the THINNEN
%       parameter is ignored.
%    2) Edge thinning: The method of thinning depends on the value
%       of THINNEN:
%       0: No edge thinning
%       1: Edge thinning with THINNENEDGESLINEAR. This method is
%          best suited for photographical images.
%       2: Edge thinning with THINNENEDGESSIMPLE. This method is
%          best suited for artificial images.
%    3) Edge selection: If optional arguments ALPHA, K1 and K2
%       are present then surround inhibition is applied using
%       these three parameters (see SURROUNDINHIBITION).
%    4) Thresholding:
%       - THRESHOLDHIGH < 0: No thresholding, but enhance the
%         brightness using the BRIGHTNESS parameter. BRIGHNESS:
%         0: No brightness enhancement
%         1: Enhance with 30%
%         2: Enhance with 60%
%       - THRESHOLDHIGH >= 0 & THRESHOLDLOW < 0: Apply
%         thresholding with threshold THRESHOLDHIGH (See THRESHOLD).
%       - THRESHOLDHIGH >= 0 & THRESHOLDLOW >= 0: Apply hysteresis
%         thresholding with TLOW = THRESHOLDLOW and THIGH =
%         THRESHOLDHIGH (see HYSTERESISTHRESHOLD).
%    IMG must be of type PImage. In the first function result, the
%    result of the operator is returned as TImage. In matrix WINNERS per
%    pixel a value i is returned if SIGMAS(i) was used at that pixel.
%
%  Examples:
%    1: dGop(img, 1, 2, -1, -1, 0);
%       Apply a dG filter with sigma=1, thinnen the edges, but do
%       not threshold the result. The result is a gradient
%       magnitude image where all nonzero locations are edges.
%    2: dGop(img, [1,2,4], 0, 0.08, 0.20, 0, 2, 4, 1);
%       Apply multiscaling to image IMG using sigmas 1,2 and 4.
%       Then perform surround inhibition with ALPHA=2, K1=4 and K2=1.
%       At last apply hysteresis thresholding with TLOW=0.08 and THIGH=0.20.
%       The result is a binary image with ones at edge locations
%       and zeros at nonedge pixels.
%
% See also: TIMAGE, PIMAGE, MULTISCALE, THINNENEDGESSIMPLE,
%           THINNENEDGESLINEAR, THRESHOLD, HYSTERESISTHRESHOLD,
%           SURROUNDINHIBITION, CANNY

% calculation of the gradient of IMG and edge thinning:
if (length(sigmas) > 1)
  % apply multiscaling. This returns the gradient (Lx,Ly) and the winners
  % info as defined in the function header. M is the gradient magnitude.
  [Lx Ly winners] = multiscale(img, sigmas);
  M = getGradientMagnitude(Lx, Ly);
else
  % get the gradient of the input image. WINNERS is one for all pixels
  % as only a single sigma, SIGMAS(1) is used.
  [Lx Ly] = gradientOfGaussian(img, sigmas(1), 1);
  winners = ones(size(Lx.img));

  if (thinnen == 2)
    % thinnen the edges using non-maximum suppression using comparison
    % with neighbouring pixels.
    M = thinnenEdgesSimple(Lx, Ly);
  elseif (thinnen == 1)
    % thinnen the edges using non-maximum suppression (use linear
    % interpolation)
    M = thinnenEdgesLinear(Lx, Ly);
  else
    % get the gradient magnitudes
    M = getGradientMagnitude(Lx, Ly);
  end
end

% apply surround inhibition if the optional arguments ALPHA, K1, K2 are present.
% This means that there are (at least) 9 function arguments. As the surround
```

```
% inhibition function expect a PImage, M is translated into a PImage Mp. After
% the call it is freed.
if (nargin >= 9)
 Mp = PImage(M);
 M = surroundInhibition(Mp, alpha, k1, k2, sigmas, winners);
 freePImage(Mp);
end

% thresholding:
if (thresholdHigh < 0)
 % translate the magnitudes image to an intensity image and
 % enhance the brightness.
 result = toIntensityImg(scaleLinear(M), brightness);
elseif (thresholdLow < 0)
 % threshold the gradient magnitudes
 result = threshold(M, thresholdHigh);
else
 % threshold the gradient magnitudes using hysteresis thresholding
 result = hysteresisThreshold(M, thresholdLow, thresholdHigh);
end;
```

---

rhperformance.m

```
function [result, E, FP, FN, VISUAL] = rhperformance(edgemap, groundtruth, sigmas, winners)
% RHPERFORMANCE measures the performance of a contour detector
%   [P, E, FP, FN, VISUAL] = RHPERFORMANCE(EDGEMAP, GROUNDTRUTH, SIGMAS, WINNERS)
%    returns the performance P of an edge detector. EDGEMAP is a binarization, a TImage,
%    containing per pixel a one if it is an edge pixel and zero otherwise. It is created
%    by a contour detector using the given list of SIGMAS. These are the scales at which
%    the contour detector looked for edges. WINNERS(x,y) should be i if SIGMAS(i) was used
%    at pixel (x,y) in the construction of the edgemap.
%    GROUNTRUTH, also a TImage with values in range [0,1], is a handdrawn image. If the drawer
%    thinks a pixel is an edge pixel then he marks such a pixel with a value of the form
%    R=accuracy/255, B=0, G=0 (in case the GROUNTRUTH is a RGB image) or accuracy/255 if the groundtruth
%    is an intensity image, where accuracy>0. Pixels with colors of another form are non-edges.
%    'accuracy' is the accuracy with which edge pixels are drawn in the groundtruth.
%    For example a value of 5 indicates, that the real edge may be located
%    anywhere in a 5 x 5 area of the pixel.
%    There are 5 function results:
%     - E: the amount of correctly detected edge pixels.
%     - FP: the amount of false positives (ie the amount of edge pixels present in the
%       edgemap, but not in the groundtruth).
%     - FN: the amount of false negatives (ie missed edges)
%     - RESULT: the actual performance, which equals E/(E+FN+FP). It is a value in
%       range [0..1]. 0 means no edge is detected correctly and 1 that all edges are
%       detected correctly.
%     - VISUAL, a color image of type TImage. Edge pixel (x,y) is marked green if it is detected
%       correctly, red if it was missed and blue if it is a false possitive.
%
%    Example: apply multiscaling and hysteresis thresholding to image 'gnu.pgm' and
%             meassure the performance, given groundtruth 'gnu_groundtruth.pgm'.
%      sigmas = [1, 2, 4];
%      groundtruth = rhloadimage('gnu_groundtruth.pgm');
%      [edgemap, winners] = dGop(rhloadimage('gnu.pgm'), sigmas, 0, 0.05, 0.2, 0);
%      performance = rhperformance(edgemap, groundtruth, sigmas, winners);
%
% See also: TIMAGE, DGOP, RHLOADIMAGE

% First it is important to define the term 'correct detection':
% An edge pixel (x,y) is detected correctly if
% - it is marked as an edge in the edgemap and it is located in the indicated inaccuracy region of
%   a pixel P closeby in the groundtruth AND
% - the distance to P does not exceed 3 times the scale at which the edge was found
%   this means it should not exceed 3*SIGMAS(WINNERS(x,y)).

% The image, in which all pixels are set to one, located within the inaccuracy region of a pixel
% in the groundtruth, is groundtruthN1.
% Also create an image edgemapN1 in which only groundtruth edge pixels remain for which an edge pixel is present
% in the edgemap in its inaccuracy region.
% The image, in which all pixels are set to one, located within the area with width and height 3*SIGMAS(WINNERS(Px,Py))
% of a pixel (Px,Py) is edgemapN2.
% groundtruthN2 is the image in which a pixel (x,y) is set to one if a groundtruth pixel is present at a distance from maximal
```

```
% 3*SIGMAS(WINNERS(x,y))/2 from (x,y).
% Now in groundtruthN = groundtruthN1 & groundtruthN2 in which all pixels that are possibly edges are marked and
% edgemapN = edgemapN1 & edgemapN2 are all edge pixels from the groundtruth that are correctly detected.
% - Now the correctly detected edges are edgemapN
% - The false positives are edgemap & ~groundtruthN
% - The false negatives are ~edgemapN & groundtruth

% Get the image field of the edgemap and create two matrices of the size of the edgemap filled with zeros:
edgemap = edgemap.img(:,:,1);
edgemapN1 = zeros(size(edgemap));
groundtruthN1 = zeros(size(edgemap));

% The groundtruth is supplied as an intensity or color image with values in range [0,1]. (This is the format
% as returned when the groundtruth is loaded with RHLOADIMAGE).
% We translate the groundtruth to an intensity image in which all nonedge pixels are set to 0 and the edge pixels are set to the
% drawing accuracy
% [1,255] of the edges.
if (size(groundtruth.img, 3) == 3)
  % The supplied groundtruth is the input image in which edge pixels are replaced by the special color R=accuracy/255, B=0, G=0.
  groundtruth = round(255*groundtruth.img(:,:,1)) .* ((groundtruth.img(:,:,1)>1e-10) & (groundtruth.img(:,:,2)<1e-10) &...
                      (groundtruth.img(:,:,3)<1e-10));
else
  % The groundtruth is an intensity image
  groundtruth = round(255*groundtruth.img(:,:,1));
end

% first find groundtruthN1 and edgemapN1.
% We walk along all possible accuracies i (ie 1..255). Then we mark all pixels using this accuracy i with one and the others with
zero
% and store the result in p (this is a simple element-wise comparison of the groundtruth with i). If i is indeed used (this means that
% the sum of all elements in p is larger than 0) then we enter the if loop.
% We can check per pixel (x,y) if a pixel is set in an area of d(x,y) x d(x,y) pixels near
% (x,y) in an image I by convolving the image with a kernel containing solely ones of size d(x,y) x d(x,y).
% Note: d is simply the accuracy function, which equals the (transformed) groundtruth.
% The idea is to assume d(x,y)=i for all pixels. Then we do the convolution of the edgemap with a kernel of size i x i containing
% solely ones.
% Afterward we set all values of pixels in the result to zero that did not have accuracy i (this can be done with a multiplication by
p)
% By or-ring all the results found using all i's we find edgemapN1.
% Using a simular method groundtruthN1 is found.
for i=1:255, % walk along all possible accuracies
  p = (groundtruth == i); % p(x,y) is one if groundtruth(x,y) equals i and 0 otherwise
  if (sum(sum(p)) > 0) % i is indeed used if the sum of the elements in matrix p is larger than 0
    kernel = ones(i); % create a convolution kernel of size i x i containing solely ones
    groundtruthN1 = groundtruthN1 | conv2(groundtruth .* p, kernel, 'same'); % extend groundtruthN1 with the result of accuracy i
    edgemapN1 = edgemapN1 | (conv2(edgemap, kernel, 'same') .* p); % extend edgemapN1 with the result of accuracy i
  end
end

% edgemapN2 and groundtruthN2 can be found using a simular method as described above, only now function d equals the
function
% 3*round(sigmas(winners(x,y))) where the results are rounded towards the neirest odd number.
% With i we walk along all values of sigma used in the construction of the edgemap.
edgemapN2 = zeros(size(edgemap)); % create a matrix filled with zeros of the same dimensions as the edgemap
groundtruthN2 = zeros(size(edgemap)); % create a matrix filled with zeros of the same dimensions as the edgemap
for i=1:length(sigmas), % walk along all sigmas used
  a = (3*round(sigmas(i))); a = a + (1-mod(a,2)); % assume the sigma used at all pixels is SIGMAS(i). So d(x,y) =
%                                                   3*round(sigmas(i)) = a the second statement on this row is used to round the
%                                                    result to the neirest odd number, so that we create a kernel of odd width.
  p = (winners == i); % p(x,y) is one if sigmas(i) was used at pixel (x,y) and 0 otherwise
  kernel = ones(a); % create a kernel containing solely ones of size a x a
  edgemapN2 = edgemapN2 | conv2(p .* edgemap, kernel, 'same'); % extend edgemapN2 with the result from i
  groundtruthN2 = groundtruthN2 | (conv2(groundtruth, kernel, 'same') .* p); % extend groundtruthN2 with the result from i
end;

% combine edgemapN1, edgemapN2 into edgemapN and groundtruthN1 and groundtruthN2 in groundtruthN:
edgemapN = edgemapN1 & edgemapN2;
groundtruthN = groundtruthN1 & groundtruthN2;

% clear variables
clear edgemapN1;
clear edgemapN2;
clear groundtruthN1;
clear groundtruthN2;
```

```
% E is the amount of correctly detected edges:
E = sum(sum(edgemapN & groundtruth));

% FP is the amount of false possitives:
FP = sum(sum(edgemap & ~groundtruthN));

% FN is the amount of false negatives:
FN = sum(sum(~edgemapN & groundtruth));

% The performance meassure now is E / (E + FP + FN)
if (E+FP+FN == 0)
 % there are no edges and you detect none -> performance is 1
 result = 1;
else
 result = E / (E + FP + FN);
end

% create an image in which the errors can be seen (see function header)
if (nargout >= 5)
 visual(:,:,1) = ~edgemapN & groundtruth;
 visual(:,:,2) = edgemapN & groundtruth;
 visual(:,:,3) = edgemap & ~groundtruthN;
 visual = TImage(visual);
end
```

---

rhloadimage.m

```
function result = rhloadimage(filename)
% RHLOADIMAGE loads an image from a file
%  RHLOADIMAGE(FILENAME) returns the image contained in file FILENAME
%    as a TImage. Each value in the result has range [0, 1]. FILENAME
%    is a character string.

% assume file is a PGM file, if not the empty string is returned
% else an image matrix.
result = readPGM(filename);

if isstr(result)
 % file is not a PGM file -> use buildin Matlab image reader

 % read image and image info
 [im,colmap] = imread(filename); % im=loaded image, colmap=color map
 info = imfinfo(filename);
 h = size(im,1); % height of image
 w = size(im,2); % width of image

 if (strcmp(info.ColorType, 'indexed'))
  % image is indexed, so translate it to RGB values using colmap.
  % colmap is a function mapping an index to a vector [R G B].
  % colmap indexes start at 1, but the
  % indexes in the image start at 0, so to get the RGB
  % value of pixel (x,y) use: RGB(x,y) = colmap(im(y,x) + 1).
  result = zeros(h,w,3); % create array for result
  for y = 1:h
   result(y,:,:) = colmap(double(im(y, :)) + 1, :);
  end
 elseif (strcmp(info.ColorType, 'grayscale'))
  % image is an intensity image

  % divide each intensity value by the amount of possible graylevels
  % minus one to get a value in range [0, 1].
  result = double(im(:,:)) / (bitshift(1, info.BitDepth) - 1);
 elseif (strcmp(info.ColorType, 'truecolor'))
  % image is an RGB image

  % We need to scale the values to range [0,1]. If the maximum value in the
  % image is less than one, assume scaling is already done otherwise
  % use the bitdepth information of the image.

  maxc = max(max(max(im))); % maximum color value
  if (maxc > 1)
```

```
    % the maximum color value is larger than 1, so scaling is needed.
    % As there are 3 color components (R, G and B) the amount of bits used per colorcomponent
    % is info.BitDepth/3, so color values are in range [0, 2^(info.BitDepth/3) - 1].
    % So scaling the color values to range [0,1] can be done by deviding them
    % by (2^(BitDepth/3) - 1):
    result = double(im) / (bitshift(1,fix(info.BitDepth/3))-1);
   else
    % maximum color value is less than 1, so assume color values are in
    % range [0.0, 1.0], the correct range.
    result = im;
   end
  else
   % error: return just one pixel
   result = 0;
  end
end

% translate the result to a TImage
result = TImage(result);

% ---------- local functions ------------------------------------

function result = readPGM(filename)
% read a PGM image file. On error the empty string is returned

[path, name, ext] = fileparts(filename); % split filename in path, name and extension
if strcmp(upper(ext), '.PGM') % check if the uppercase of the extension is '.PGM'
 % file is a pgm file -> read it
 % A pgm file has the following format:
 % a one line header ('P5')
 % a comment (one line)
 % one line with 2 unsigned integers: the image width and height
 % one line with a single unsigned integer: the maximal color value allowed (only 255 supported for now)
 % the data: a height x width matrix
 f = fopen(filename, 'r'); % open file read only
 if ~(strcmp(fgetl(f), 'P5')) % read first line, the file header, and check if it is 'P5'
  % header is not 'P5', so file is not a pgm file
  result = '';
  return;
 end
 fgetl(f); % skip line containing the comment
 a = fscanf(f, '%u%u%u'); % read width, height and maximum color value, ie a = (width,height,maxcol)
 if (a(3) ~= 255) % if maximum color is not 255 then error
  result = '';
  return;
 end

 % read the image data. The image must be rotated by 90 degrees, to make it stand upright
 % (this can be done by the single quote operator). Then scale the values to range [0,1] by
 % dividing the result by the maximum color (this is stored in a(3)):
 result = fread(f, [a(1),a(2)])' / a(3);

 fclose(f); % close the file
else
 % not a pgm file
 result = '';
end;
```

---

saveimg.m

```
function saveimg(img, filename)
% SAVEIMG save an image to a file
%   SAVEIMG(IMG, FILENAME) saves the image IMG to given file.
%   IMG must have type TImage, FILENAME is a string.
%
%   Example: saveimg(TImage(zeros(64,64)), 'blackbox.jpg')
%
% See also: TIMAGE, RHLOADIMAGE

if (min(min(min(img.img))) < 0) | (max(max(max(img.img))) > 1)
 % if image data isn't in range [0, 1] already then scale it
 img = scaleLinear(img);
```

```
end;
img = img.img; % get image field

[path, name, ext] = fileparts(filename); % split filename in path, name and extension
if strcmp(upper(ext), '.PGM') % check if the extension is '.pgm'
 % save file as pgm file

 % A pgm file has the following format:
 % a one line header ('P5')
 % a comment (one line)
 % one line with 2 unsigned integers: the image width and height
 % one line with a single unsigned integer: the maximal color value allowed (only 255 supported for now)
 % the data: a height x width matrix

 % get dimensions of the image
 [h w ncolors] = size(img);

 % pgm files are graylevel -> check if IMG is this
 if (ncolors ~= 1)
  error('Image must be graylevel when saved as a PGM file');
 end

 % translate graylevel doubles to 8 bit unsigned integers and rotate the image by 90 degrees
 img = uint8(fix(img * 255))';

 % write the PGM file (see format description above):
 f = fopen(filename, 'w');
 fprintf(f, 'P5\n');
 fprintf(f, '# PGM file generated by PGM writer for Matlab by R. Hof\n');
 fprintf(f, '%u %u\n%u\n', w, h, 255);
 fwrite(f, img, 'uint8');
 fclose(f);
else
 % use Matlab image writer
 imwrite(img, filename);
end
```

---

rhimage.m

```
function result=rhimage(img, h)
% RHIMAGE displays an image
%  H = RHIMAGE(IMG) displays the given image IMG in the current figure.
%    If no figure is present then one is created with the same size as
%    the image. A handle to the image object is returned.
%    IMG must be of type TIMAGE
%    The (color) intensities are scaled down to range [0, 1]
%    before displaying.
%  H = RHIMAGE(IMG, '') display given image IMG of type TImage in a
%    new figure and return the figure's handle.
%  H = RHIMAGE(IMG, H) display given image IMG of type TImage in
%    axes H, returns H.
%
% See also: TIMAGE

img = rhmkimgviewable(img); % make the image IMG displayable.

if (isempty(get(0, 'CurrentFigure'))) | ((nargin == 2) & isstr(h))
 % no figure exists or second call form -> create one with the
 % size of the image and center it.
 [h w C] = size(img.img); % get height, width and amount of color components
 scrsize = get(0, 'ScreenSize'); % retreive the screen size
 fig=figure('Position', [(scrsize(3) - w) / 2,(scrsize(4) - h) / 2,w-1,h-1], ...
                        'MenuBar', 'none'); % create the figure centered on the screen
 set(gca, 'Position', [0,0,1,1]); % make image axes use the whole figure
elseif (nargin == 2)
 axes(h); % make h the current axes
end

% display the image without axes in the current figure
result = image(img.img); % display the image
set(gca, 'Visible', 'off'); % hide the horizontal and vertical axes
```

rhmkimgviewable.m

```
function result = rhmkimgviewable(img)
% RHMKIMGVIEWABLE transforms an image into one that can be displayed
%  RHMKIMGVIEWABLE(IMG)
%    scales the image IMG down to range [0, 1] and translates it
%    into a RGB image. IMG must be of type TIMAGE.
%
% see also: TIMAGE, RHIMAGE, SCALELINEAR

if (min(min(min(img.img))) < 0) | (max(max(max(img.img))) > 1)
 % if image data isn't in range [0, 1] already then scale it
 result = scaleLinear(img); % scale all color components to range [0,1]
else
 % image data is already in range [0, 1]
 result = img;
end;

C = size(result.img, 3); % get amount of color components
if (C == 1)
 % intensity image -> RGB by setting R,G and B to the same intensity.
 result.img(:,:,2) = result.img(:,:,1);
 result.img(:,:,3) = result.img(:,:,1);
end
```

toIntensityImg.m

```
function result = toIntensityImg(img, brightness)
% TOINTENSITYIMG translates an image to an intensity image
%  TOINTENSITYIMG(IMG, BRIGHTNESS) translates image IMG to an intensity
%    image. Brightness can be 0 (no brightness enhancement, 1 =
%    medium, 2 = bright). values in IMG and result should be in range
%    [0,1]. Both IMG and function result must be variables of type
%    TImage.
%
% See also: TIMAGE

[h w ncolors] = size(img.img); % height, width, amount of color components in the image.
result(h, w) = 0; % clear a matrix of size h x w to zero.

% calculate Intensity = (R+G+B)/3 and enhance the brightness.
for c = 1:ncolors,
 b = brightness * 0.3;
 result(:,:) = result(:,:) + img.img(:,:,c) .* (1.0 - b) + b;
end

% last step in RGB -> intensity: a division by 3
if (ncolors == 3)
 result(:,:) = result(:,:) / 3;
end

% translate the result to a TImage
result = TImage(result);
```

scaleLinear.m

```
function result = scaleLinear(img)
% SCALELINEAR scales the values of an image to range [0, 1]
%  SCALELINEAR(IMG) scales all color components of an image IMG
%    to range [0, 1] by means of linear scaling. IMG must be
%    a TImage. The result is a TImage as well.
%
% See also: TIMAGE

[h w C] = size(img.img); % get height, width and amount of color components

for c = 1:C,
```

```
  % get minimum and maximum color value of component c in result
  minc = min(min(img.img(:,:,c)));
  maxc = max(max(img.img(:,:,c)));

  % scale values in the result to range 0..1
  if (maxc - minc ~= 0)
    % there are multiple colors -> map colors lineary to range [0, 1]
    result(:,:,c) = ((img.img(:,:,c) - minc) / (maxc - minc));
  else
    % if only a single value is used, then set all pixels in the result to black
    result(h,w,c) = 0;
  end;
end

% translate the result to a TImage
result = TImage(result);
```

---

convolve.m

```
function result = convolve(img, k);
% CONVOLVE applies convolution
%   CONVOLVE(IMG, K) convolves an image IMG of type PImage with a
%     convolution kernel K (a matrix). The result is a TImage.
%
%   Example:
%     convolve(PImage(rhloadimage('filename')), [-1:1;-1:1;-1:1]))
%
% See also: TIMAGE, PIMAGE, RHLOADIMAGE


%%---------------------- preprocessing:----------------------

% get width, height and amount of color components in the image
[h, w, ncolors] = size(getfield(TImage(img), 'img'));

% The input image and convolution kernel must have the same size. This is done by
% centering both image and filter in a matrix, whose size is the the sum of the
% size of the image and kernel, NH x NW. NH and NW must be even, so NH and NW are
% rounded towards the neirest even number.
% Note: for more efficiency use max(filterWidth, w) instead of filterWidth+w (and idem for
% the height. Speedups are considerable (approximately a factor 2!). This results
% in artefacts near the image borders though.
[filterHeight, filterWidth] = size(k); % size of the kernel is filterHeight x filterWidth
%  nh = (filterHeight + h) + mod(filterHeight + h, 2);
%  nw = (filterWidth + w) + mod(filterWidth + w, 2);
nh = (max(filterHeight,h)) + mod(max(filterHeight, h), 2);
nw = (max(filterWidth,w)) + mod(max(filterWidth, w), 2);


% The input image is tranformed to a new image of size NH x NW by centering
% it in a matrix of this size and reflecting the image at its borders.
% Then the FFT of this new image is calculated and stored in the PImage IMG by RHIMGFFT.
% (So only img.fimg is affected, not img.img !)
rhimgfft(img, nw, nh);

% get the TImage structure stored in IMG
in = TImage(img);

% convolution using FFT:

% We need to center both kernel and image in a matrix of
% size NH x NW. For the image this is done by RHIMGFFT, for the kernel
% in this function.
% Calculate the position where their topleft point should come.
cix = fix((nw - w)/2); % (cix, ciy) is the location of the top left point
ciy = fix((nh - h)/2); % of the image in the NH x NW matrix.
ckx = round((nw - filterWidth)/2); % idem (ckx, cky) for the kernel
cky = round((nh - filterHeight)/2);

% now center the kernel in a matrix, kc, filled with zeros of
% size NH x NW.
kc(nh, nw) = 0;
```

```
% center the kernel
kc(cky+1:cky+filterHeight,ckx+1:ckx+filterWidth) = k(:,:);

% calculate the fft of the kernel
K = fft2(kc, nh, nw);

%% ---------- convolve the input image with the kernel ----------
% convolution is done per color component
for c = 1:ncolors,
  % do convolution in the frequency domain
  KI = K(:,:) .* in.fimg(:,:,c);
  % translate the result back to image domain and swap quadrants using fftshift
  resultUnclipped = real(fftshift(ifft2(KI)));

  % the image of size h x w is in the center of the image of size
  % NH x NW, so extract it
  result(:,:,c) = resultUnclipped(ciy+1:ciy+h, cix+1:cix+w);
end

% return the result as a TImage
result = TImage(result);
```

---

normalize.m

```
function result = normalize(M)
% NORMALIZE normalizes a matrix with the L1-norm
%   NORMALIZE(M) normalizes matrix M with the L1-norm,
%     ie result = M / sum(sum(abs(M)));

result = M / sum(sum(abs(M)));
```

---

getGradientMagnitude.m

```
function result = getGradientMagnitude(Lx, Ly)
% GETGRADIENTMAGNITUDE finds the gradient magnitude
%   GETGRADIENTMAGNITUDE(LX, LY) calculates the magnitude
%     sqrt(LX^2 + LY^2). LX, LY and function result are of type
%     TImage.
%
% See also: TIMAGE, GRADIENTOFGAUSSIAN

result = TImage(sqrt(Lx.img .^ 2 + Ly.img .^ 2));
```

PImage.m

```
function result = PImage(img, varname)
% PIMAGE creation of a variable of type PImage.
%   PIMAGE(TIMG) creates a global, unique identifier and stores
%     TIMG in this variable. The name of this variable is returned
%     as a character string. TIMG must be of type TImage. If it is
%     not then the function tries to convert it to a TImage.
%   PIMAGE(TIMG, VARNAME) creates a global identifier with name
%     VARNAME and stores TIMG in this variable. VARNAME is a
%     character string, which is also returned as function result.
%     TIMG must be of type TImage. If it is not then the function
%     tries to convert it to a TImage.
%
%   Note: A PImage variable can be seen as a pointer to a TImage
%   variable. You need to give up the memory claimed by the PImage
%   with a call to freePImage.
%
%   Examples:
%     PImage(rhloadimage('filename')) creates a PImage from given
%       image file.
%     PImage(rhloadimage('filename'), 'animg') creates
%       a global variable with name 'animg' and stores a picture
```

```
%      with given filename in this variable.
%
%  See also: TIMAGE, RHLOADIMAGE, FREEPIMAGE

if (nargin == 1)
 % VARNAME isn't present -> create an unique global variable

 % try to define variables of form TImageXXXXX, where XXXXX is a
 % random number globally, till we get one which isn't defined yet.
 defined = 1; % is current variable defined ?
 while (defined),
  % create a random name of form TImageXXXX and put it in result:
  varname = ['TImage', int2str(fix(100000 * rand))];
  % define this name globally:
  eval(['global ', varname]);
  % test if result <> [], or in other words if it's a new variable,
  % if not we reenter the while-loop:
  defined = ~isempty(eval(varname));
 end
else
 % VARNAME is specified -> declare it global
 eval(['global ', varname]);
end

if ~isTImage(img)
 % if the input image isn't a TImage then turn it into one
 img = TImage(img);
end

% assign this structure to the identifier with name as stored in 'result'.
eval([varname, '=img;']);

% return the name of the global variable in which the TImage is stored
result = varname;
```

TImage.m

```
function result = TImage(img);
% TIMAGE creates a variable of type TImage
%  TIMAGE(RGBIMG) creates a structure with fields 'img' and 'fimg'
%   RGBIMG is a matrix of dimensions H x W x 3,
%   where H is the height and W the width of the image. RGBIMG(:,:,1),
%   RGBIMG(:,:,2) and RGBIMG(:,:,3) are the red, green and blue
%   components of the image. if RGBIMG(:,:,1)==RGBIMG(:,:,2)==RGBIMG(:,:,3)
%   then only RGBIMG(:,:,1) is stored in the 'img' field else RGBIMG.
%   fimg is a placeholder for the FFT of the image, which is
%   set to -1 meaning not calculated yet.
%  TIMAGE(INTENSITYIMG) creates a structure with fields 'img' and 'fimg'
%   INTENSITYIMG is a matrix of dimensions H x W,
%   where H is the height and W the width of the image.
%   INTENSITYIMG is stored in the 'img' field of the result structure.
%   'fimg' is a placeholder for the FFT of the image, which is
%   set to -1 meaning not calculated yet.
%  TIMAGE(PIMG) retreives the TImage structure stored in the
%   global variable with name PIMG. PIMG must be of type PImage.
%   The TImage structure has one of the forms described above.
%
% See also: PIMAGE, RHIMGFFT

if (isstr(img))
 % call form TIMAGE(PIMG)
 eval(['global ', img]); % define the variable with name 'img' global
 result = eval(img); % retreive the structure at the variable img
 if (~isTImage(result)) % check if this structure is a TImage, if not raise an error
  error('Illegal callform');
 end
elseif (isnumeric(img))
 % call forms TIMAGE(RGBIMG) and TIMAGE(INTENSITYIMG)

 if (length(size(img)) > 3)
  % 2D or 3D numeric matrix expected
  error('Illegal callform');
```

```
  else
    % ncolors is the amount of colorcomponent 1 or 3. If the input
    % image is an INTENSITYIMG then this is 1 else it's 3.
    ncolors = size(img, 3);

    if (ncolors == 1)
      % input image is a gray level image
      result.img = img;
    elseif (ncolors == 3)
      % input image is a RGB image.
      if (img(:,:,1) == img(:,:,2)) & (img(:,:,2) == img(:,:,3))
                        % Check if the colorcomponents per pixel are all equal if so
                        % only store the first color component.
                        result.img = img(:,:,1);
      else
                        % store the RGB image in the img field.
                        result.img = img;
      end
    else
      error('Illegal callform');
    end

    % the fft of the input image hasn't been calculated yet
    result.fimg = -1;
  end
else
  error('Illegal callform');
end
```

---

isTImage.m

```
function result = isTImage(img)
% ISTIMAGE is true for TImages
%   ISTIMAGE(IMG) returns true if IMG is of type TIMAGE
%
% See also: TIMAGE

% an image is a TImage if is is a structure with fields img and fimg:
result = 0;
if (isstruct(img))
  result = ((isfield(img, 'img')) & (isfield(img, 'fimg')));
end
```

---

freePImage.m

```
function freePImage(img);
% FREEPIMAGE give up memory used by a PIMAGE
%   FREEPIMAGE(IMG) releases memory used by the variable
%   which name is stored in IMG. IMG must be a
%   character string.
%
% See also: PIMAGE

eval(['clear global ', img]);
```

---

rhfix.m

```
function result = rhfix(nr, varargin)
% RHFIX floor that takes calculational errors into account
%   RHFIX(NR, [PRECISION]) returns round(nr) if
%     the fraction of the number is at least PRECISION
%     else the floor of the number is taken. The default
%     value of PRECISION is 0.9999.
%     Because of calculational errors the default fix may cause
%     problems: log(8) / log(sqrt(2)) is exactly 6.
%     Because of calculational errors,
```

```
%    the outcome of this formula is 5.999999..., taking the
%    floor of such a number yields 5. rhfix returns 6, the
%    correct answer.
%
% See also: FLOOR

if (length(varargin) > 0)
  % the precision is specified, use it
  precision = varargin{1};
else
  % use the default precision
  precision = 0.9999;
end

if (nr - floor(nr) >= precision)
  % the fraction of the number is at least 0.9999...,
  % so round it upwards.
  result = ceil(nr);
else
  % else use the default fix function
  result = fix(nr);
end
```

---

rhsign.m

```
function result=rhsign(n, e)
% RHSIGN sign function insensitive to computational noise
%   RHSIGN(N, [E]) returns the sign (-1, 0 or 1) of a number. The
%   problem with numerical analysis is the finite machine
%   precision. Testing for zero on such numbers is bound to fail,
%   because of round off errors during computations. The function returns
%   -1 if N < -E, 1 if N > E and zero otherwise. E should be a very
%   small number (default = 1E-10).
%
% See also: SIGN

if (nargin < 2)
  % E is not specified, use default
  e = 1E-10;
end

% find the sign:
result = sign(n) .* (abs(n) > e);
```

---

mkvalid.m

```
function result = mkvalid(M, v)
% MKVALID maps nonfinite values to another (finite) value
%   MKVALID(M,V) takes care that each nonfinite value (ie NaN, Inf or
%     -Inf) in numeric matrix M is mapped to number V.

% calculate the amount of elements, Msize, in M by multiplying its dimensions:
s = size(M);
Msize = 1;
for i=1:length(s),
  Msize = Msize * s(i);
end

% idea: if a value is not finite than the calculation of the index
% in the code below puts the value at the index of the nonfinite
% value to v else
% the v is put at index 1. That's why index 1 must be
% dealth with seperately.
result = M;
result((0:Msize-1) .* ~isfinite(M(1:Msize)) + 1) = v;
if (isfinite(M(1)))
  result(1) = M(1);
end
```

rhimgfft.m

```
function result=rhimgfft(img, nw, nh)
% RHIMGFFT updates the FFT of an image
%  RHIMGFFT(IMG, NW, NH, FIMG) calculates the fft of IMG,
%    a variable of type PImage.
%    The image is first expanded to size NH x NW by means of
%    reflection. NH and NW must be even numbers.
%    img^.fimg is the previous result of this function. If
%    its -1 or its size differs from NH x NW then the FFT
%    is recalculated. As function result the PImage IMG is
%    returned. This is a helper function used by CONVOLVE.
%
% See also PIMAGE, CONVOLVE

% in = img^, ie the structure stored in variable with name img.
in = TImage(img);

% The need for reflection:
% The reason for reflecting the image to size NH x NW is that
% convolution is done in the frequency domain, for which kernel and image
% must have the same size.

% Note on NW and NH: there seems to be a problem if NW or NH
% is odd. To solve the problem, NW and NH should rounded towards the neirest
% number devidable by 2.

h = size(in.img,1); % height of image
w = size(in.img,2); % width of image
fimgxf = size(in.fimg,2); % width and height of fimg
fimgyf = size(in.fimg,1);

if ((in.fimg == -1) | (fimgxf ~= nw) | (fimgyf ~= nh))
  % if FIMG unknown or width/height changed then recalculate the fft
  % of IMG

  % reflect the image to get an image of size NH x NW
  rin = reflect(in, nw, nh);

  ncolors = size(rin.img, 3); % the third dimension is one if the
                  % input image is a graylevel image else
                  % it is three.

  % calculate the FFT of IMG per color component
  in.fimg = zeros(nh, nw, ncolors);
  for color=1:ncolors,
    in.fimg(:,:,color) = fft2(rin.img(:,:,color), nh, nw);
  end

  % store the result in the global variable with name as stored in IMG.
  PImage(in, img);
end

% return the name of the global variable in which the result is stored.
result = img;
```

reflect.m

```
function result = reflect(img, nw, nh)
% REFLECT enlarges an image by reflecting it at its borders
%  REFLECT(IMG, NW, NH) centers the image part of IMG in a matrix of size
%    NWxNHx? and fills up the regions of the matrix that are not
%    covered by the image by reflecting the image at its borders.
%    If NH x NW is less or equal than the size of IMG then IMG is returned.
%    IMG and result must be of type TImage.
%
% See also: TIMAGE

% idea: a matrix of size NH x NW is created. A virtual grid with cell size
% w * h is placed on top of it. We put the original image in the
```

```
% center, this is grid cell (0,0). There are both xn cells to the
% left and right of it and yn above and beneath. For each cell we
% calculate wether to just copy the image there or mirror it (in x
% and or y direction). This is represented by the variables xdir
% and ydir. Of course at the edges only parts of the cells there
% fall within the result matrix. Here the image copy should be
% properly clipped. This is done by local function calcpos.

[h, w, c] = size(img.img); % get height, width, colormodel (1 = gray, 3 = RGB)

if (((nw == w) & (nh == h)) | ((nw < w) | (nh < h)))
 % image already has correct size
 result = img;
 return;
end

% clear result matrix
result(nh, nw, c) = 0;

xn = ceil((ceil(nw/w)-1) / 2); % nr of full/half copies of image
                % left and right of the center image.
yn = ceil((ceil(nh/h)-1) / 2); % idem for above and beneed.

cx = fix((nw - w) / 2); % pos where image should come in matrix
cy = fix((nh - h) / 2); % (cx, cy) is the topleft pos of cell (0,0)

% walk through all grid cells
for y = -yn:yn,
 for x = -xn:xn,

  bx = cx + x * w; % pos of current image copy
  by = cy + y * h;

  % xdir: copy direction. -1 copy from right to left, 1 from left to
  % right from input image. ydir is simular.
  xdir = (1 - mod(abs(x), 2)) * 2 - 1;
  ydir = (1 - mod(abs(y), 2)) * 2 - 1;

  % calculate where the copy of the input image should start and end in
  % both x and y direction
  [rxstart, rxend, ixstart, ixend] = calcpos(xdir, bx, w, nw);
  [rystart, ryend, iystart, iyend] = calcpos(ydir, by, h, nh);

  % copy / reflect the image at the previously calculated location
  for ncolors = 1:c,
   result(rystart:ryend, rxstart:rxend, ncolors) = ...
                img.img(iystart:ydir:iyend, ixstart:xdir:ixend, ncolors);
  end
 end
end

% transform the result to a TImage
result = TImage(result);


%---------- local functions

function [rstart, rend, istart, iend]  = calcpos(dir, b, size, nsize)
% calculates where in the result matrix (rstart:rend) a copy of the
% image (istart: iend) should be placed. It should be called twice,
% once for the x and one for the y direction. dir is the copy
% direction (backwards = -1, forwards = 1), b is the top/left
% location of the copy in the result matrix, size is the width or
% height of the original data, nsize is the width or height of the
% result matrix.
if (dir < 0)
 if (b < 0)
  rstart = 1;
  rend = b + size;
  istart = size + b;
  iend = 1;
 elseif (b + size - 1 >= nsize)
  rstart = b + 1;
  rend = nsize;
  istart = size;
```

```
    iend = (b + size - 1) - (nsize - 1) + 1;
  else
    rstart = b + 1;
    rend = (b+1) + size - 1;
    istart = size;
    iend = 1;
  end
else
  if (b < 0)
    rstart = 1;
    rend = b + size;
    istart = abs(b) + 1;
    iend = size;
  elseif (b + size - 1 >= nsize)
    rstart = b+1;
    rend = nsize;
    istart = 1;
    iend = nsize - (b+1) + 1;
  else
    rstart = b + 1;
    rend = (b+1) + size - 1;
    istart = 1;
    iend = size;
  end
end
```

---

maximg.m

```
function [result, indexes] = maximg(a, b)
% MAXIMG the pixelwise maximum of two images
%   [RESULT, INDEXES] = MAXIMG(A, B)
%     returns the pixelwise maximum of images A and B in RESULT
%     and in indexes per pixel a one if the maximum came from A
%     and a two if it came from B.
%     A, B and function result must be of type TImage. Images
%     A and B must have the same dimensions. INDEXES is a matrix
%     with the same dimensions as RESULT.img.
%
% see also: TIMAGE

[h w ncolors] = size(a.img); % get the dimensions of A

% determine the pixelwise maximum of A and B per colorcomponent c
for c = 1:ncolors,
  % put image A behind image B. The result is a matrix of size hxwx2
  tmp = reshape([a.img(:,:,c), b.img(:,:,c)], h, w, 2);

  % determine the maxima in the direction of the third dimension
  % using the max function.
  [result(:,:,c), indexes(:,:,c)] = max(tmp, [], 3);
end
clear tmp;

% turn the result into a TImage
result = TImage(result);
```

---

mkLogList.m

```
function result = mkLogList(low, high, k)
% MKLOGLIST creates a list with values on a logaritmic scale
%   MKLOGLIST(LOW, HIGH, K) creates a list with the values
%     [LOW * K^0, .., LOW * K^n, HIGH] where n is the largest
%     natural number such that LOW * K^n < HIGH.

% test if the parameters are valid. HIGH must be at least
% LOW and K, LOW must be positive.
if (low > high)
  error('LOW must be less or equal to SIGMAHIGH');
end
```

```
if (k <= 0)
  error('K must be positive');
end
if (low <= 0)
  error('LOW must be positive');
end

% calculate the largest whole number n with
% K^n LOW < HIGH. This can be done
% by solving p in the equation k^p LOW == HIGH.
% n now equals the floor of p with one exception. If
% p is a natural number then this would give K^n LOW ==
% HIGH, so 1 needs to be substracted from n.
% If the fraction of p, ie p-floor(p), is zero then p is a
% natural number. Because of calculational errors one needs
% to check for zero with some error marge (ie |p-floor(p)| < err),
% where err is a very small number, for example 1e-10. Instead
% of floor rhfix is used (see rhfix for the problems with floor).
p = log(high / low) / log(k);
n = rhfix(p);
if (abs(n - p) < 1e-10)
  n = n - 1;
end

% the list of values to return is:
% [LOW, K LOW, K^2 LOW, .., K^n LOW, HIGH].
% This list is stored in the result result. We round the values to 10 digit precision.
result = round([(k .^ (0:n)) .* low, high]*1e10)/1e10;
```