Matematicko-fyzikální fakulta

Univezity Karlovy

Diplomová práce

Igor Omboš

Detekce meteorů na širokoúhlém CCD snímku

Meteorite Detection System for Widefield CCDs

Kabinet software a výuky informatiky

Department of Software and Computer Science Education

Vedoucí diplomové práce: RNDr. Josef Pelikán

Studijní program: Informatika, Softwarové systémy

Praha, 2007

Poděkování

Petru Kubánkovi, poradci této práce, za velkou ochotu, nadšení pro věc a trpělivost.

Mým rodičům za ještě větší ochotu, nesmírné nadšení pro mě a trpělivost přesahující moji veškerou chápavost.

A všem, kteří mi v průběhu mého studia pomohli.

Contents

Název práce:	Detekce meteorů na širokoúhlém CCD snímku
Autor:	Igor Gomboš
Katedra (ústav):	Kabinet software a výuky informatiky (KSVI)
Vedoucí diplomové práce:	RNDr. Josef Pelikán
E-mail vedoucího:	josef.pelikan@mff.cuni.cz

Abstrakt: BOOTES je česko-španělský robotický systém pro automatickou správu teleskopů umístěných v Ondřejove a v pohoří Sierra Nevada.
Cílem diplomové práce je navrhnout a implementovat nástroj pro detekci stop meteoritů a otestovat ho na širokouhlých astrografických snímcích vysokého rozlišení pořízených systémem BOOTES.
Důraz je kladen především na rychlost analýzy, protože software musí snímky zpracovávat v reálném čase (doba pro běh je maximálně 20 až 30 sekund). Důležitou vlastností je i otevřenost a snadná rozšiřitelnost zdrojového kódu.
Prezentované řešení nabízí na výběr ze tří různých metod detekce. První využívá síly standardní Houghovy transformace pro detekci čar. Druhá je založena na Cannyho hranovém detektoru a poslední metoda je jednodušší verzí výplňové segmentace.
Klíčová slova: meteority, detekce čar, Houghova transformace

Title:	Meteorite Detection System for Widefield CCDs
Author:	Igor Gomboš
Department:	Department of Software and Computer Science Education
Supervisor:	RNDr. Josef Pelikán
Supervisor's e-mail address:	josef.pelikan@mff.cuni.cz

Abstract: *Lúthien* is a software package designated to process astrographic images created by *BOOTES*, a Czech-Spanish robotic telescope system.
As an extension to the RTS2 library, its purpose is to process large-sized astrographic images taken by a widefield camera. Due to high storage cost, it is necessary to treat the input data in real time (20 to 30 seconds being the maximum). Apart from that, the software is open and easily extensible.
The proposed solution offers three detection methods to choose from. The first one implements the Standard Hough transform. The other two are based on the Canny edge detector and simple region growing, respectively.
Keywords: meteorites, line detection, Hough transform

Chapter 1

Preface

Lúthien Tinúviel
Tinúviel was a name given to her by Beren. It literally means 'daughter of the starry twilight', which signifies 'nightingale'. She is described as the Morning Star of the Elves.
Silmarillion

In general, human beings are *very* curious. They see something unknown and they stare at it first. Then they come closer, smell it, and try to touch it. And finally, *the thing* inevitably ends in the mouth and gets eaten.

Unfortunately, it is not that simple with stars. We cannot go and touch them, we cannot even smell them (smelloscope of Dr. Farnworth does not count). So the voyers (called scientists) look and take pictures. They analyze the imagery and search for some interesting objects. Manual search is very inefficient, slow and talent-wasting. And here comes *Lúthien* to help.

The main motivation of the thesis is to find and implement suitable semi-automatic or automatic computer vision algorithms to speed up the routine process of the meteorite search. The software should be fast (real-time in terms of seconds) and easily extensible for later needs.

This chapter summarizes contents and formulates goals of this work.

1.1 Thesis Structure

The work is divided into two parts - 7 chapters and 4 appendices. The chapters present details of the ideas standing behind the implementation façade. On the other hand, the appendices are of rather practical nature.

Right now you are reading the first chapter which discusses the motivation, structure and goals of the thesis. Some necessary evil of definitions is brought by the second chapter, along with a brief presentation of various pre-processing methods; basic morphology techniques come next. Chapter number three reviews line extraction approaches in general, putting more accent on the beloved Hough transform. The fourth chapter describes the detection procedures implemented in *Lúthien*. Details about software and hardware environment can be found in the chapter with number five in its name. Chapter number six summarizes the results of practical experiments, including some example images. Conclusion and the seventh chapter are the same, to the letter.

Appendices A and B are supposed to serve as User's and Developer's documentation of the produced software. Resources used during the development are presented in Appendix C. Finally, Appendix D is a listing of the default configuration file.

1.2 Problem Formulation and Goals

The main goal of this thesis is to deliver a software product dedicated to detect meteorite trails present in high-resolution astrography images (circa 16MPix, for more details see Section 5.1). The images are taken by a widefield CCD camera of the BOOTES system (see Section 5.3). Moreover, this software should be fast enough to process the pictures within the scope of seconds (30 seconds being the maximum). This is because the BOOTES system telescope takes continuously one image per minute and memory cost of storing the pictures for later processing would be too high.

The thesis is also supposed to review suitable image-processing line detection techniques. The application should implement more than one of them in order to perform some comparison testing. The target platform for *Lúthien*, as we named the software, is GNU Linux. User manual and developer documentation are, as required, part of the eventual work. The applicability of the tool should be demonstrated on real data.

Chapter 2

Used Digital Image Processing Techniques

Using interdisciplinary knowledge from psychology, biology, physics, mathematics, and even art, *digital image processing* has become very successful in number of areas of human activity; ranging from medical diagnostics and industrial inspection, nuclear medicine and astronomical observations, through industry and astronomy X-ray imaging, lithography, microscopy, to radars and biological imaging.

In principle, digital image processing covers two main application areas - *image enhancement* to improve 'information readability' of images to a human being and *processing* and *interpretation* of image data done by autonomous machine perception systems.

Image processing can be divided into some general stages such as image acquisition, image enhancement, image restoration, compression, morphological processing, segmentation, representation and description. All of them involve a great multitude of various procedures and methods which are not possible nor requisite to present in the scope of this thesis. Following chapter represents a brief overview of some basic image processing definitions and approaches, with more attention paid to techniques directly used in the practical implementation of *Lúthien*.

2.1 Definitions

2.1.1 Discrete Image

Definition 1 *A* continuous image *is a mapping* $\Omega = [0, x] \times [0, y]$ *to a co-domain* $\mathbb{R}^n$*:*

$$f : \mathbb{R}^2 \supset \Omega \to \mathbb{R}^n,$$

where $n \in \mathbb{N}$, $x \in \mathbb{R}$ and $y \in \mathbb{R}$. Domain Ω is called image domain *or* image plane, *x is the* width *of the image, y is the* height *of the image. In case $n = 1$, we talk about a* continuous greyscale image.

The output data of most sensors is continuous voltage waveform. However, to create a digital image (also called a *discrete image*), we hate to convert these continuous data into digital form, representable in a computer. The conversion involves discretization of the domain Ω - *sampling* (digitizing the coordinate values) and discretization of the co-domain - *quantization* (digitizing the amplitude; typically into 256 levels per color channel).

Definition 2 *A* discrete image *is a mapping $\chi = [0, x] \times [0, y]$to a co-domain $\mathbb{Z}^n$:*

$$\mathcal{P} : \mathbb{N}^2 \supset \chi \to \mathbb{Z}^n,$$

where $n \in \mathbb{N}$, $x \in \mathbb{N}$ and $y \in \mathbb{N}$, x represents the width *of the image, y is the* height *of the image. In case $n = 1$, we talk about a* discrete greyscale image.

In further chapters only discrete images are considered.

2.1.2 Convolution

Convolution is an integral that expresses the amount of overlap of one function as it is shifted over another function. It therefore 'blends' one function with another. Abstractly, a convolution is defined as the product of two functions f and g (objects in the algebra of Schwartz functions in R^n), after one of them is reversed and shifted.

Definition 3 *Convolution of two functions f and g over a finite range $[0, t]$ is given by*

$$(f * g)(t) = \int_t^0 f(\tau) g(t - \tau)\, d\tau,$$

where the symbol $f * g$ (occasionally also written as $f \otimes g$) denotes convolution of f and g.

Convolution is more often taken over an infinite range,

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau = \int_{-\infty}^{\infty} g(\tau) f(t - \tau)\, d\tau$$

Typically, one of the functions is taken to be a fixed filter impulse response, and is known as a *kernel.*

For discrete functions, a discrete version of the convolution (called *normal convolution*) is employed. It is given by

$$(f * g)(m) = \sum_n f(n)g(m-n)$$

Image processing usually uses a 2-D version of convolution which has the form

$$g(i,j) = \sum_{(m,n)\in O} \sum h(i-m, j-n)\, f(m,n) \tag{2.1}$$

The resulting value in the output image pixel $g(i,j)$ is calculated as a *linear combination* of values in the local neighborhood O of the pixel $f(i,j)$ in the input image. The contribution of the pixels is weighted by coefficients of kernel h which is called a *convolution mask*.

Convolution is used to implement many different operators, particularly spatial filters and feature detectors. Examples include Gaussian smoothing (see Section 2.2.3.1) and the Sobel edge detector (see 2.2.3.2).

2.1.3 Gaussian distribution

The *normal distribution*, also called the *Gaussian distribution*, is an important family of continuous probability distributions. Each member of the family may be defined by two parameters, location and scale: the *mean* ('average', μ) and *variance* ('variability', σ^2), respectively.

Due to the central limit theorem, the normal distribution plays a significant role of a model of quantitative phenomena in the natural and behavioral sciences. Many psychological measurements and physical phenomena (like noise) can be approximated by the normal distribution.

Definition 4 *The continuous probability density function of the normal distribution is the* Gaussian function

$$\varphi_{\mu,\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}, \quad x \in \mathbb{R},$$

where $\sigma > 0$ *is the* standard deviation *and the real parameter* μ *is the* expected value.

The Gaussian distribution in 1-D has the form

$$\varphi_{\mu,\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{x^2}{2\sigma^2}\right\}, \quad x \in \mathbb{R}$$

In 2-D, an *isotropic* (i.e., circularly symmetric) Gaussian has the form

$$\varphi_{\mu,\sigma^2}(x,y) = \frac{1}{\sigma^2 2\pi} \exp\left\{-\frac{x^2+y^2}{2\sigma^2}\right\} \tag{2.2}$$

$$\textit{or} \quad \varphi_{\mu,\sigma^2}(x,y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{x^2+y^2}{2\sigma^2}\right\},$$

where $x, y \in \mathbb{R}$.

2.2 Image Pre-processing

Pre-processing is a common name for operations with images at the lowest level of abstraction. The aim of digital image pre-processing is to increase both the interpretability and the accuracy of the treated data. The irrelevant information is suppressed, but the information content is never increased. There are four basic groups of pre-processing methods: pixel (point) operations, geometric transformations, local neighbourhood-using methods and methods that work with global information. The ones relevant to this work are presented in this chapter.

2.2.1 Image Arithmetic

Image arithmetic is the most simple form of image processing. One of the standard *arithmetic operations* or a *logical operator* is applied to one, two or more images. The value of the output pixel depends only on the values of the corresponding input pixels. In static scenes, for instance, adding subsequent images in random noise reduction, or subtracting two subsequent images in motion detection, may be helpful. To selectively process only a part of an image, a binary mask is often used. The logical operators are incorporated in binary image processing mostly.

Some basic arithmetic operations:

- ▷ *pixel addition* – $g(i,j) = f_1(i,j) + f_2(i,j)$
- ▷ *pixel subtraction* – $g(i,j) = f_1(i,j) - f_2(i,j)$ or $g(i,j) = |f_1(i,j) - f_2(i,j)|$
- ▷ *pixel multiplication and scaling* – $g(i,j) = f_1(i,j) \times f_2(i,j)$
- ▷ *pixel division* – $g(i,j) = f_1(i,j) \div f_2(i,j)$
- ▷ *blending* – $g(i,j) = P \times f_1(i,j) + (1-P) \times f_2(i,j)$,

where $g(i,j)$ is the output pixel value at position (i,j), $f_1(i,j)$ and $f_2(i,j)$ are the input pixel values at position (i,j), and P is the blending ratio.

The truthtables for logical operators are as follows:

	A	0	0	1	1
logical AND	B	0	1	0	1
	Q	0	0	0	1

	A	0	0	1	1
logical NAND	B	0	1	0	1
	Q	1	1	1	0

	A	0	0	1	1
logical OR	B	0	1	0	1
	Q	0	1	1	1

	A	0	0	1	1
logical NOR	B	0	1	0	1
	Q	1	0	0	0

	A	0	0	1	1
logical XOR	B	0	1	0	1
	Q	0	1	1	0

	A	0	0	1	1
logical NXOR	B	0	1	0	1
	Q	1	0	0	1

invert/logical NOT	A	0	1
	Q	1	0

2.2.2 Point Operations

Point operations (also called *grayscale transformations*) do not depend on the position of the input pixel and do not take the grey value configuration in the neighborhood into account. The greyscale is modified globally.

We can describe a point operation as a mapping ϕ of the original pixel value $f(x,y)$ into a new pixel value $g(x,y)$ as follows

$$\phi: \ f(x,y) \longrightarrow g(x,y) = \phi(f(x,y))$$

ϕ can be defined in an ad-hoc manner (e.g., for gamma correction, or thresholding), or it can be computed from the input image (as for histogram equalization).

There are two basic types of point operations - *affine grayscale transformations* and *nonlinear grayscale transformations.* Affine transformations have simple structure $g(x,y) = a \cdot f(x,y) + b$. Contrast enhancement, contrast attenuation, brightening, darkening, greyscale reversion are of such nature. In case $b = 0$, they are also called *linear grayscale transformations.* Nonlinear transformations include thresholding, logarithmic dynamic compression, gamma correction and histogram equalization, to name a few.

Thresholding is a simple, yet powerful method for segmentation. It can be described by the transformation

$$g(x,y) = \phi(f(x,y)) := \begin{cases} 1 & for\ f(x,y) \geq T \\ 0 & else \end{cases},$$

where T is the *threshold* value, $g(x,y) = 1$ for image elements and $g(x,y) = 0$ for background elements (or vice versa). See Figure 2.1 for an example. *Global thresholding* uses a single threshold for the whole image (successful only under special circumstances), hence some advanced variants like *adaptive thresholding* (also *variable thresholding*), *optimal thresholding or multi-thresholding [52]* were proposed.

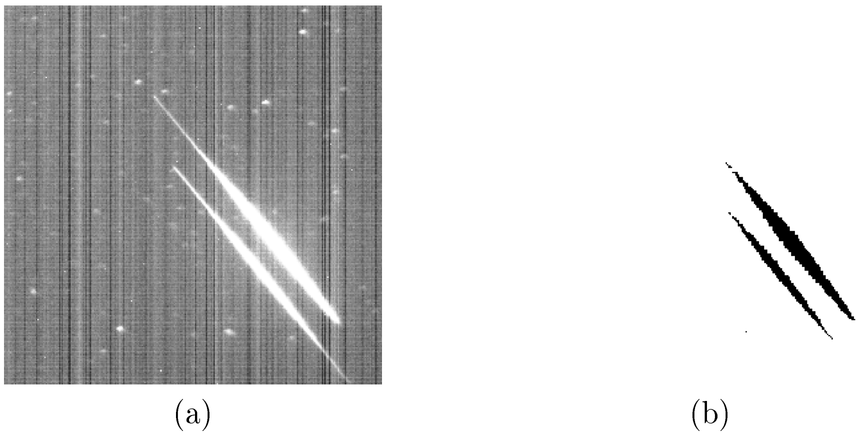


Figure 2.1: Thresholding: (a) original image, (b) simple threshold applied (result inverted).

Histogram (also *intensity histogram*) is a graph, which specifies relative occurence frequency of pixel intensity values within an image. Spatial context does not matter, since any permutation of the image pixels gives the same histogram. The histogram can be treated as a discrete probability density function. *Histogram equalization* is a monotonic transformation $g = \tau(f)$ modifying the histogram so that pixel brightness levels are distributed equally over the whole scale. The image contrast enhancement is usually acompanied by a dramatic improvement in the subjective image quality. The actual implementation is shown in Algorithm 1 and an example of equalized histogram is demonstrated in Figure 2.2.

2.2.3 Local Pre-processing

Filtration operations (or *filtering*) are pre-processing methods that work with the values of a small neighborhood of the input pixel and the corresponding values of a subimage with the same dimensions as the neighborhood. The subimage is referred to as a *kernel*, *window*, *filter*, *mask*, or *template* and its values are called *coefficients*.

Algorithm 1 Histogram Equalization

```
1: **Input:** A $W \times H$ discrete image, L is the total of possible brightness levels in the image.
2: Initialize an array $Arr$ of length $L$ with zeroes.
3: Create the image histogram.
4: Create the cumulative image histogram $CHr$
       $$CH[0] = Arr[0]$$
       $$CH[i] = CH[i-1] + Arr[i] \qquad i = 1, 2, \ldots, L-1$$
5: Compute the new brightness values transform table
       $$Tab[i] = round(\frac{L-1}{WH} CH[i])$$
6: The output image graylevels are
       $$g_{out} = Tab[g_{inp}]$$
   .
```

Filtering can take place in the *frequency domain* (Fourier transform) or in the *spatial domain*, directly on the pixels of the image. In this section, only spatial filtering is discussed. For frequency domain approaches overview, see [23].

We distinguish two basic groups of local pre-processing methods - *smoothing* and *gradient operators*. *Smoothing* techniques suppress small fluctuations in the image (typically noise), but blur edges at the same time. *Gradient operators* extract local changes in the intensity function (edge is a group of pixels where the intensity function changes significantly).

Linear pre-processing transformations calculate the output value as a linear combination of pixel values in a close neighborhood of the input pixel. The contribution of the neighboring pixels can be desribed as discrete convolution by Equation 2.1 .

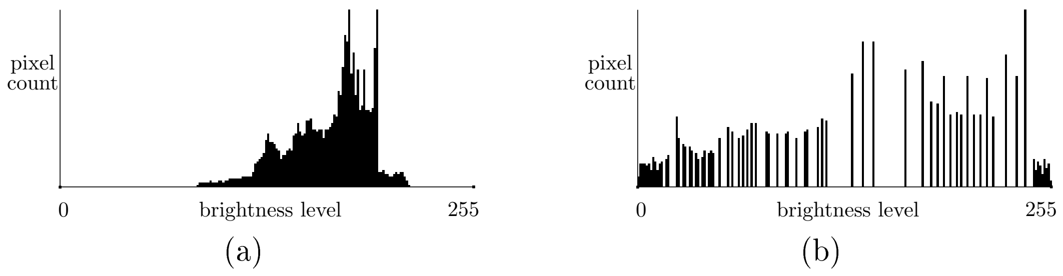


Figure 2.2: Histogram equalization: (a) original histogram, (b) equalized histogram.

2.2.3.1 Image Smoothing

Smoothing filters eliminate small details (*blurring*) in the image and reduce noise using information redundancy in the image data. The smoothing operation is, in principle, averaging the brightness values of neighboring input pixels. Impulse noise and thin stripe degradations can be effectively eliminated by smoothing. Such filters are called *averaging* or *lowpass filters.*

A *box filter* is a spatial averaging filter, whose all coefficients are equal. It computes the standard average of the pixels under the subwindow. For a 3×3 neighborhood, the convolution mask is

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Example of box filtration is depicted in Figure 2.3.

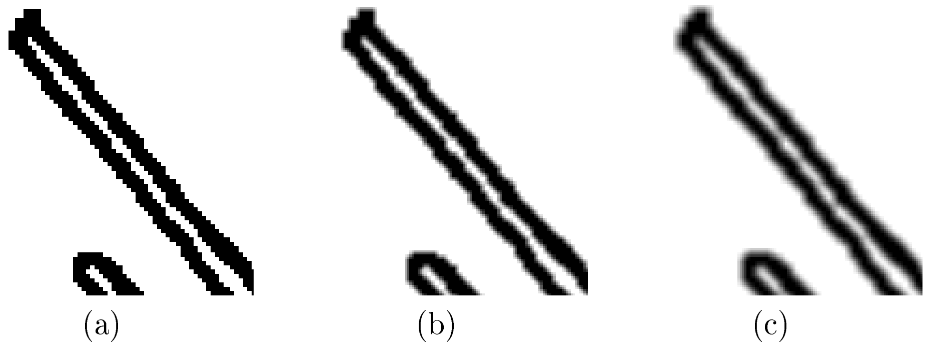


Figure 2.3: Box filtering: (a) original image, (b) filtered by a 3×3 box filter, (c) filtered by a 5×5 box filter.

The center of the *weighted average* convolution kernel is sometimes given increased significance to approximate the properties of a Gaussian probability distribution noise in more a accurate manner. We can create Gaussian convolution kernels according to the Gaussian distribution formula 2.2. Hence, the 2-D *Gaussian smoothing operator* $G(x, y)$ (also *Gaussian*) is given by

$$G(x,y) = \frac{1}{\sigma^2 2\pi} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \quad \textit{or} \quad G(x,y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right), \tag{2.3}$$

where σ is the standard deviation, the only parameter of the filter. It defines size of the neighborhood on which the filter operates. Some example Gaussian kernels (two 3×3 Gaussians and one 5×5 Gaussian) are defined as follows

$$h = \frac{1}{10}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad h = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad h = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Alternative smooting approaches, involving rotating mask averaging, median filtering, order statistics filtering, are presented in [52].

2.2.3.2 Feature Detectors

As mentioned before, an edge can be interpreted as an abrupt change (discontinuity) in the image intensity function behaviour. Changes of continuous functions are described using *derivatives*. As image function is usually 2-D function, partial derivatives and gradient are used. *Gradient* shows the direction of the largest growth of the function. An edge at a particular pixel position has two properties - *magnitude* (size of the gradient) and *direction* (perpendicular to the gradient direction).

The gradient of $f(x, y)$ at coordinates (x, y) is defined as a column two-dimensional vector

$$\nabla f = \begin{bmatrix} grad_x \\ grad_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

The gradient magnitude and the gradient direction ϕ are given by

$$mag(\nabla f) = \sqrt{grad_x^2 + grad_y^2} = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

$$\phi = arg(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}),$$

where $arg(x, y)$ is the angle from the x-axis to the point (x, y), measured in radians.

Because of high computational costs, it si common to approximate the gradient magnitude by

$$mag(\nabla f) \approx\mid grad_x \mid + \mid grad_y \mid \tag{2.4}$$

First-order derivatives are usually approximated by differences

$$\frac{\partial f}{\partial x} = f(x+1, y) - f(x, y) \quad and \quad \frac{\partial f}{\partial y} = f(x, y+1) - f(x, y)$$

Examples of the first-order derivative approximating operators will be presented in the following paragraphs.

One of the oldest and simplest is *Roberts Cross operator* with convolution masks

$$h_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad h_2 = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Only four pixel values are used. Thus, the output is easy to compute, but highly sensitive to noise.

Prewitt operator is one of the *compass operators,* which are able to compute the gradient direction, as well. The first two convolution masks of the Prewitt operator are defined as follows (others are created by simple rotation)

$$h_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \qquad h_2 = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}$$

Sobel operator is a simple, yet very popular pre-processing edge detector. Its first three convolution masks (again, others created by rotation) are given by

$$h_1 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad h_2 = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \quad h_3 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Sometimes (usually for horizontal and vertical edge detection) only h_1 and h_3 masks are used. In such case the magnitude can be computed as $\sqrt{x^2 + y^2}$ or approximated by $\mid x \mid + \mid y \mid$, where x is the response to the h_1 mask and y is the response to the $h_2$convolution mask (see Figure 2.4). The gradient direction is given by $\tan^{-1}(\frac{y}{x})$.

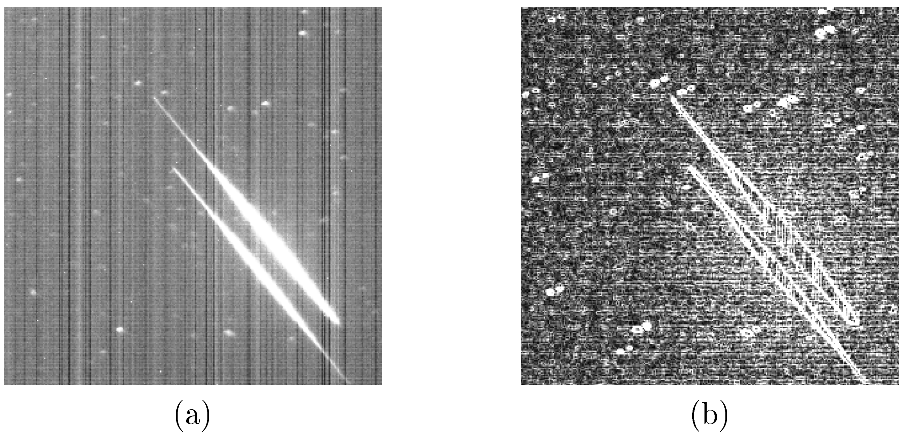

(a) (b)

Figure 2.4: Sobel edge detection: (a) original image, (b) sobel detection result (masks h_1 and h_3 applied).

Other first-order operators are *Kirsch operator* and *Robinson operator* [52].

To enhance fine details, second-order derivatives are sometimes used. They act more aggressively in enhancing sharp changes (thin lines, isolated points, but also noise). Moreover, they produce thinner edges in an image than the first-order derivatives. Their disadvantage is double-response to step changes. While the first derivative has an extremum at the position of an edge (abrupt brightness function change), the second derivative is equal to zero at the same position (see Figure 2.5). Yet it is much easier to find a zero-crossing than the exact extremum coordinates.

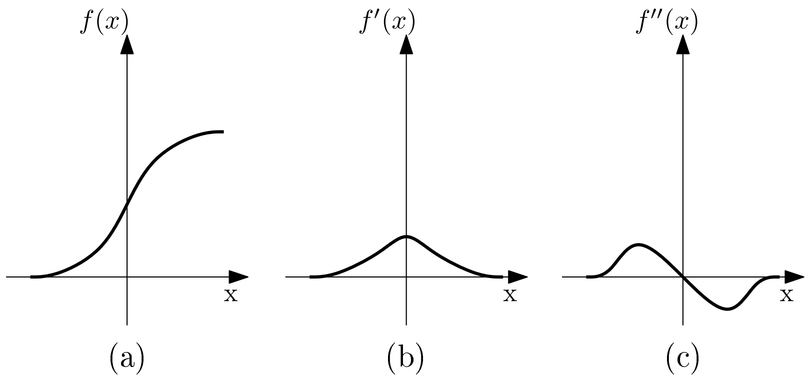


Figure 2.5: First and second derivatives of 1-D function: (a) original function $f(x)$, (b) its first derivative, (c) second derivative of $f(x)$. Notice the zero-crossing in (c).

A very popular second-order derivative approximating operator is the *Laplace operator* (also *Laplacian*) defined as

$$\nabla^2 f = \frac{\partial^2 f}{\partial^2 x^2} + \frac{\partial^2 f}{\partial^2 y^2}$$

Similarly to the first-order derivatives, the second-order derivatives are approximated by differences

$$\frac{\partial^2 f}{\partial^2 x^2} = f(x-1,y) - 2f(x,y) + f(x+1,y) \quad \textit{and} \quad \frac{\partial^2 f}{\partial^2 y^2} = f(x,y-1) - 2f(x,y) + f(x,y+1)$$

Masks used for practical implementations are

$$h = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad h = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

To compute the second derivative robustly (remember its high sensitivity to noise), the image is smoothed first. In practice, Gaussian smoothing operator is used. Applying the Laplacian operator ∇^2 (computes the second derivative) on a gaussian pre-smoothed image is called *Laplacian of Gaussian* (abbrev. *LoG,* or $\nabla^2 G$)

$$\nabla^2 \left[G(x, y, \sigma) \star f(x, y) \right]$$

Using linearity of the operators and substitution we get the equation for an LoG convolution mask

$$h(x, y) = c \left(\frac{x^2 + y^2 - \sigma^2}{\sigma^4} \right) \exp \left(-\frac{x^2 + y^2}{2\sigma^2} \right),$$

where c is a normalizing multiplicative constant. Examples of discrete 5×5 and 7×7 LoG approximation follow

$$h = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix} \quad h = \begin{bmatrix} 0 & 0 & -1 & -1 & -1 & 0 & 0 \\ 0 & -1 & -3 & -3 & -3 & -1 & 0 \\ -1 & -3 & 0 & 7 & 0 & -3 & -1 \\ -1 & -3 & 7 & 24 & 7 & -3 & -1 \\ -1 & -3 & 0 & 7 & 0 & -3 & -1 \\ 0 & -1 & -3 & -3 & -3 & -1 & 0 \\ 0 & 0 & -1 & -1 & -1 & 0 & 0 \end{bmatrix}$$

Because of its shape, the inverted $\nabla^2 G$ is often referred to as *Mexican hat.* The LoG operator is often approximated by difference of Gaussians with different standard deviation values, called *difference of Gaussians* (*DoG*).

For the zero-detection phase a true zero crossing detector implementation is necessary. For example, we can use a 2×2 moving window. We assign the 'edge label' to the upper left pixel, if LoG values of both polarities (positive and negative) appear in the window simultaneously.

2.2.3.3 Canny Edge Detector

In [7] John F. Canny came with an edge detector intended to be optimal according to three basic criteria. First, no important edges should be missed and no false positives should be detected. Second, the edge points should be localized as well as possible (the distance between the detected and actual edge location is minimal). And the third criterion says that no multiple responses to a single edge are allowed.

Canny took advantage of the fact that convolving an image using a symmetric 2 dimensional Gaussian and differentiating the result perpendicular to the edge direction (in the gradient direction) creates an effective and simple directional operator

$$\frac{\partial G}{\partial \mathbf{n}} = \mathbf{n}.\nabla G = G_n,$$

where G is a 2-D Gaussian, G_n the resulting operator and $\mathbf{n}$ the direction perpendicular to the edge. We can approximate the direction using smoothed gradient direction as

$$\mathbf{n} = \frac{\nabla (G \star f)}{\mid \nabla (G \star f) \mid}$$

The local maximum of the image f convolved with G_n in the direction $\mathbf{n}$ then gives us the edge location

$$\frac{\partial}{\partial \mathbf{n}} G_n \star f = 0$$

Using substitution for G_n we get

$$\frac{\partial^2}{\partial^2 \mathbf{n}} G \star f = 0 \qquad (2.5)$$

This approach to find local maxima in the direction perpendicular to the edge is called *non-maximal suppression*. It is used to suppress any pixel value not considered to be an edge pixel, leaving just thin edge lines (Figure 2.6(b)).

Convolution and derivative are associative operations. In Equation 2.5, advantage of this fact is taken. The order of applied operations is changed, convolving the image with the Gaussian G *before* computing the second-derivative (using the Sobel operator, for instance). The magnitude of the gradient (also the strength of the edge) can be found as

$$\mid G_n \star f \mid = \mid \nabla (G \star f) \mid$$

To get the edge pixels, the second-derivative output is usually thresholded. Using single threshold value would induce the 'streaking edges problem' (breaking up of the edge contours), as the output fluctuates above and below the threshold. *Thresholding with hysteresis* is used to eliminate the 'streaking'. Pixels with edge strength above the *high threshold* are immediately declared edge pixels (called *edgels*). Pixels with edge strength above the *low threshold* are considered edgels if they are connected to a high-threshold edge pixel (Figure 2.6(c)).

All in all, the effect of the Canny operator is determined by three parameters - the smoothing Gaussian mask width (*scale* of the Canny detector), and the values of the high and the low thresholds. Increasing the width of the Gaussian makes the detector less sensitive to noise, but some of the fine detail in the image may be lost and the localization error can increase. Setting the upper tracking threshold to a high value

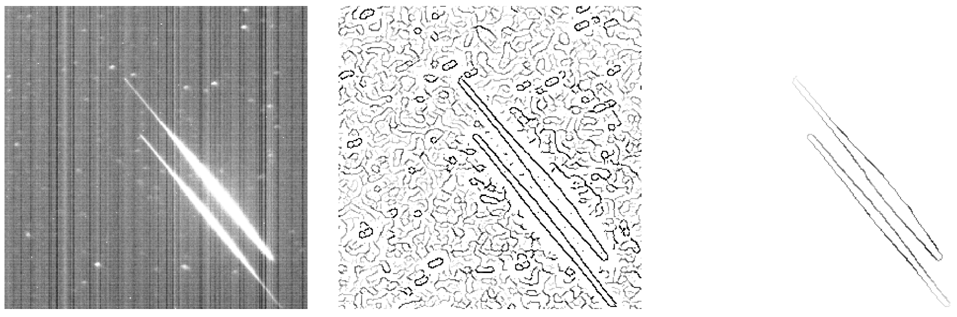

(a) (b) (c)

Figure 2.6: Canny edge detector: (a) original image, (b) non-maximal suppression result, (c) thresholding with hysteresis eliminates weak edge points.

and the lower threshold to a low value usually returns good results. Too high values of the low threshold cause noisy edges to break up, while too low values of the upper threshold increase the number of spurious edge fragments in the detector output.

Different scales of the Canny detector yield significantly different results. To solve the problem of finding the correct scale, Canny proposed a *feature synthesis* method. The detection is repeated multiple times at different scales and important edges not detected at smaller scales are accumulated into a cumulative edge map.

The Canny edge detection process is briefly described in Algorithm 2.

Algorithm 2 Canny Edge Detection

```
1: Convolve an image $f$ with a Gaussian of scale $\rho$.
2: For each pixel estimate local edge normal directions $\mathbf{n}$ using
       $$\mathbf{n} = \frac{\nabla (G \star f)}{| \nabla (G \star f) |}$$
3: Find the edge locations using non-maximal suppression.
4: Compute the edge magnitudes.
5: Threshold edges with hysteresis.
6: Repeat steps (1) through (5) for ascending values of the standard deviation $\rho$.
7: Aggregate the final information using the 'feature synthesis' approach.
```

Because of its complexity, full implementation of Canny's edge detector is unusual. Common implementations involve steps 2 - 6 of the algorithm only.

2.3 Morphology

Mathematical morphology is a theoretical model for digital images based on mathematical concepts from set theory, lattice theory and topology using shift-invariant (translation invariant) operators. The morphological operators are widely used for the image analysis, including edge detection, noise removal, image enhancement and image segmentation. Originally developed for binary images, mathematical morphology was later extended for grayscale images and multi-band images.

In a binary image, the pixels can be divided into two groups - *foreground* and *background* pixels. We can look at the image as a set of 2-D (n-dimensional, in general) Euclidean coordinates of all the foreground pixels in the image. For a grayscale image, the intensity value represents the height above a base plane. Hence, the grayscale image represents a surface in 3-D Euclidean space. Binary morphology can be seen as a special case of grayscale morphology, in which the input image has only two graylevels - 0 and 1, respectively.

The two most basic operations in mathematical morphology are *erosion* and *dilation*. Conceptually, the structuring element is translated to various pixel positions of the input image, and the intersection between the translated kernel coordinates and the input image coordinates is examined.

The *structuring element* (also known as a *kernel*) is a set of point coordinates, although often represented as a binary image. In comparison with the input image, it is usually much smaller (typically 3×3 pixels) and its coordinate origin is often in its centre instead of in one of the corners. A morphological operator is defined by its structuring element and the applied set operator.

Let A denote a binary image and B denote a structuring element, both sets in $\mathbb{Z}^2$. The basic morphology operations are defined as follows.

Erosion of A by B (denoted $A \ominus B$) is defined by

$$A \ominus B = \{z \mid (B)_z \subseteq A\}$$

For each *foreground* pixel of the input image ('input pixel'), the structuring element is superimposed on top of the input image so that the origin of the structuring element coincides with the input pixel position. The input pixel is left as a foreground pixel only in case all the input pixels coinciding with the structuring element pixels are foreground pixels. Otherwise the input pixel is marked as a background pixel.

The erosion operator applied on a binary image shrinks (erodes away) the boundaries of foreground regions in the image. Hence, areas of foreground pixels are reduced in size, while background holes within those areas become larger.

Dilation of A by B (denoted $A \oplus B$) is defined by

$$A \oplus B = \left\{z \mid (\hat{B})_z \cap A \neq \emptyset\right\}$$

where $\hat{B}$ stands for the *reflection* of B (also *rational set*, *symmetrical set* or *trans-*

pose), given by $\hat{B} = \{w \mid w = -b,\ for\ b \in B\}$.

For each *background* pixel of the input image ('input pixel') the structuring element is superimposed on top of the input image so that the origin of the structuring element coincides with the input pixel position. The input pixel is marked as a foreground pixel in case at least one structuring element pixel coincides with an input foreground pixel underneath (see Figure 2.7 for an example). Otherwise the input pixel stays a background pixel.

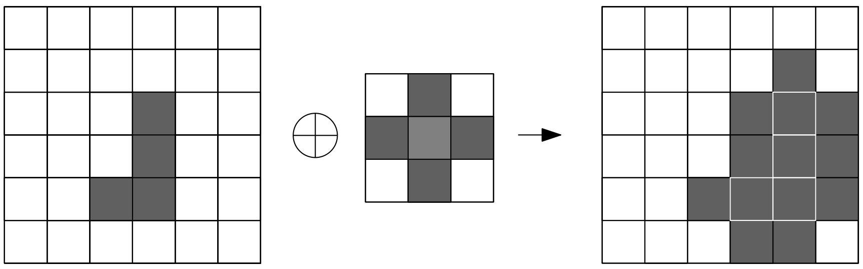


Figure 2.7: Image dilation using 4-continuity structure element (in the middle).

The dilation operator gradually enlarges the foreground pixel regions, while holes within those areas become smaller or dissapear completely.

Dilation and erosion are dual morphology operations of each other with respect to set reflection and complementation, i.e., eroding foreground pixels is equivalent to dilating the background pixels. That is,

$$(A \ominus B)^c = A^c \oplus B,$$

where A^cstands for the *complement* of A, defined as $A^c = \{w \mid w \notin A\}$. But neither erosion nor dilation is an invertible operation.

Virtually all other mathematical morphology operators can be derived by using erosion and dilation along with set operators, such as intersection and union. Some of the most important are opening, closing, hit-or-miss transformation and skeletonization.

Opening A by B (denoted $A \circ B$) is defined as

$$A \circ B = (A \ominus B) \oplus B$$

The erosion of A by B is followed by a dilation of the resulting structure by B. Opening represents a basic morphological tool for small objects (noise) and holes removal.

Closing A by B (denoted $A \bullet B$) is defined as

$$A \bullet B = (A \oplus B) \ominus B$$

The dilation of A by B, followed by erosion of the resulting structure by B. See Figure 2.8 for an example.

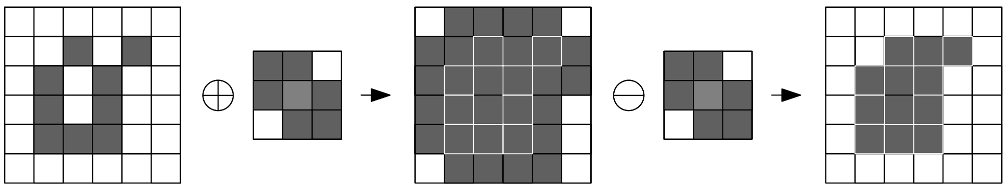


Figure 2.8: Image closing (dilation followed by erosion) using asymetrical structure element. Notice how the hole in the input structure disappeared (got 'closed').

Like erosion and dilation, opening and closing are dual morphology operations of each other with respect to set reflection and complementation, too. That is,

$$(A \bullet B)^c = A^c \circ \hat{B}$$

The *hit-or-miss transformation* is a morphological operator designated to find groups of pixels with certain shape properties, such as border points, or corners. A pair of disjoint sets $B = (B_1, B_2)$ is used as the structuring element (called a *composite structuring element*). A pixel p is marked a foreground pixel, if the B_1 part of the composite structuring element is contained in A, and the part B_2 is contained in A^c. Thus, the hit-or-miss transformation (denoted $A \odot B$) is defined by

$$A \odot B = \{p : \ B_1 \subset A \ and \ B_2 \subset A^c\}$$

The transformation actually provides a binary matching between the structuring element and the original image. We can define it using operations of erosion and dilation as well

$$A \otimes B = (A \ominus B_1) \cap (A^c \ominus B_2) = (A \ominus B_1) \setminus (A \oplus \hat{B}_2)$$

Skeletonization is a process of reducing foreground regions in a binary image to a *skeleton*, a remnant largely preserving some basic topological and size characteristics of the original shape (like extent and connectivity), while deleting most of the original foreground pixels. Originally, the idea of skeleton was proposed by Harry Blum, who illustrated it using the 'grassfire scenario'. Let a region A be a part of $\mathbb{R}^2$. A grassfire starts on the entire boundary of A at the same time and advances towards the region interior with constant speed. The skeleton $S(A)$ is then the set of points, where two or more firefronts meet. The notion of *a skeleton* as a simple and compact shape representation is intuitive, the exact definition can be found in [23]. Two example skeletons are presented in Figure 2.9.

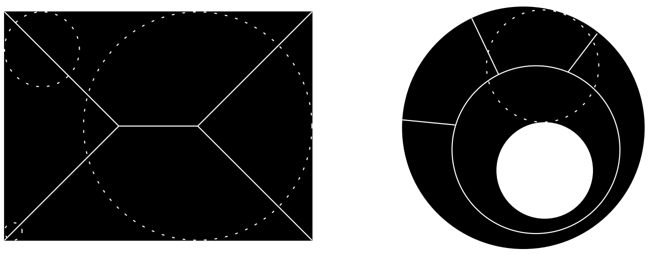


Figure 2.9: Skeletons of a rectangle and a circular object with hole.

Morphological *thinning* is one way to produce a skeleton. Successive erosion eliminates pixels from the boundary (while preserving the end points of line segments) until no more thinning is possible, leaving the approximation of the actual skeleton.

The thinning of a set A by a structuring element $B = (B_1, B_2)$ (denoted $A \oslash B$) can be defined in terms of the *hit-or-miss transform as*

$$A \oslash B = A \setminus (A \otimes B) \tag{2.6}$$

Hence, a part of the foreground boundary is eroded by the set difference operation. A more useful method for thinning A symmetrically is based on a sequence of structuring elements

$$\{B\} = \left\{B^1, B^2, B^3, \ldots, B^n\right\},$$

where B^i is a rotated version of B^{i-1}. Using this concept, we now define thinning by a sequence of structuring elements as

$$A \oslash \{B\} = ((\ldots((A \oslash B^1) \oslash B^2) \ldots) \oslash B^n)$$

The process is to thin A by one pass with B^1, then thin the result with one pass of B^2, and so on, until A is thinned by one pass of B^n. The entire process is repeated until no further changes occur. Each individual thinning pass is performed using Equation 2.6.

One of the structuring element sequences often used in practice is called the *Golay alphabet* [22]. The basic 3×3 structuring element L of the Golay alphabet is defined as

$$L_1 = \begin{bmatrix} 0 & 0 & 0 \\ * & 1 & * \\ 1 & 1 & 1 \end{bmatrix} \quad L_2 = \begin{bmatrix} * & 0 & 0 \\ 1 & 1 & 0 \\ * & 1 & * \end{bmatrix} \quad L_3 = \begin{bmatrix} 1 & * & 0 \\ 1 & 1 & 0 \\ 1 & * & 0 \end{bmatrix} \quad \ldots \quad ,$$

where the B_1 element is composed of 1 values, zeroes belong to the element B_2, and pixels at asterisk $*$ positions are insignificant in the matching process. The remaining five matrices are given by simple rotation. The thinned result consists only of 1-pixel wide lines and isolated points.

Chapter 3

Lineal Features Extraction

Finding *lineal features* (or *line detection*) in an image is a classical image processing problem and has been studied for many years. It is a basic, yet very important preprocessing step in many object recognition and scene analysis procedures, because lines can be used to describe many artificial (man-made) objects and represent a mathematically simple intermediate level primitive, at the same time.

3.1 Method Overview

There have been several approaches how to extract lineal primitives, all of which have their particular advantages and disadvantages. A brief overview of the most important categories of line detection methods is presented in this section.

3.1.1 Classic Approaches

Probably the most widely used method for lineal features extraction relies on *edge linking* and *segmentation*. Commonly, lines are viewed as extended or contiguous edges. Consequently, the basic idea is to find local edge pixels using some low-level processing (like gradient computation using some adequate mask, e.g., Sobel or orthogonal masks by Frei and Chen [21]). Then they are linked into contours on the basis of their neighbourhood and similar gradient magnitude and direction values. Finally, the contours are joined into relatively straight pieces.

The Nevatia-Babu line finder [60] along with works of Zhou et al. [78] and Nalwa and Pauchon [57] represent classic examples of this approach. Etemadi's method [19] finds chains of edgels using the Marr-Hildreth edge detector and separates the chains into pieces that are symmetric about their centroid. Consequently, it attempts to interconnect these into longer segments. A local approach using block-based line segment detection can be found in [45]. Lee and Kweon [44] developed a six-step algorithm consisting of edge detection, edge scanning, edge normalization, line-blob extraction, line-feature computation and line linking. The *stick growing* method by Nelson [59] incorporates matching measure for a line segment with explicit lineal and end-stop terms, its parallelization possibilities are discussed in [30]. The algorithm starts with small *sticks* at high-gradient starting points found by the *hill climbing* method (stick growing), then their tips are extended until the end-stop criteria are reached. Zucker et al. [79] used a relaxation process to group edges into lines; Kanazawa and Kanatani [40] proposed an asymptotic approximation to fit a model of a line to an edge segment. Eichel and Delp [62] created a sequential model to link edge pixels based on Markov random fields. This work was extended to multiresolution approaches by Cook and Delp [13, 14].

The main problem with edge (or edgel) linking approaches is that they have a strong tendency to be sensitive to the output of the edge detection. They enhance noise and may generate dense edge maps, which makes successive processing more difficult. Furthermore, they usually have trouble bridging gaps. The segmentation process can also be unstable, particularly if there is any bumpiness in the pattern. Some of these problems can be solved by using grouping techniques [50] and multiresolution representations [72].

3.1.2 Hough Transform-based Methods

The second methods parameterize potential lines in edges detected in the scene and then base the detection in the parameter space. The *Hough transform* [17] has been widely used for detecting lines in (mostly binary) images using this approach. Local edges vote for all possible lines they are consistent with. The votes are summed up to determine what lines are actually present. A verification phase may also follow. The main setbacks of this approach are computational complexity, higher storage requirements, and lack of locality. The method is rather expensive to implement, because every edge pixel must vote for all the lines it is consistent with. This can present a large figure, depending on the desired resolution and on how the segment space is parameterized. The method is nonlocal and bridges gaps well, but may combine unrelated data. Hence, extensive post-processing is often needed. Since HT is a crucial element in one of the *Lúthien* methods (see Chapter 4), it is presented in more detail in Section 3.2.

3.1.3 Alternative Techniques

The third approach of lineal feature detection utilizes the gradient direction to partition the image into a group of support regions, each of which will afterwards be associated with a single feature (due to Burns et al. [6]). A least-squares fitting procedure is used to fit a line segment to every region. This method can detect low-contrast features, but the segmentation can be unstable. Equally, features can rather easily be broken up by local perturbations, i.e., we face the gap problem again.

There are also other, less common approaches. Statistical method by Mansouri et al. [53] proposes a hypothesize-and-test algorithm to find line segments of a given length by hypothesizing their existence based on local information, and attempting to verify that hypothesis statistically on the basis of a digital model of an ideal segment edge. Work of Mattavelli and Noel [54] is based on combinatorial optimization of partitioning an inconsistent linear system, which is constructed from the coordinates of all contour points in the image as coefficients and the line parameters as unknowns. Kass et al. [51] presented a method involving an optimization process for fitting a geometric structure to data on a basis of an energy optimization procedure. Liu et al. [49] proposed a special spatial characteristic model for describing line structures in an image.

3.2 Hough Transform

The *Hough Transform* (abbrev. HT, pronounced /hΛf/) [26, 43], also called the *standard Hough transform* (SHT), was first proposed by Paul Hough [28, 34]. It is a popular method for detecting image patterns like straight lines or circles and has been generalized to extract any arbitrary shape at given orientation and scale in both binary and greyscale images.

Initially, it was used in machine analysis of bubble chamber photographs [75]. The Hough transform was patented as *U.S. Patent 3,069,654* in 1962 with the name "*Method and Means for Recognizing Complex Patterns*" [28]. This patent introduces a *slope-intercept* parametrization for straight lines. The *rho-theta* parametrization universally used today was brought by Duda and Hart in [17]. For more information about HT for *line detection*, see Section 3.2.1.

HT is essentially a *voting* process, in which each point belonging to the pattern votes for all the possible patterns passing through that point. The votes are accumulated in an *accumulator array* and the pattern receiving the maximum vote is taken to be the desired pattern. Accuracy of the transform has been discussed in [68, 5, 61]. Hardware implementations are described in [27] and [11]. Industrial applications of the transform are discussed in [69].

The main *advantages* of HT are its robustness to noise in the image and discontinuities in the pattern (gaps, occlusions), even in a complicated background. The *disadvantages* of the SHT are its demand for a tremendous amount of computing power and high memory costs. Both requirements increase linearly with the resolution, at which the parameters are determined. Moreover, digitizing and quantization errors sometimes influence the accumulation of the peaks, e.g., multiple and sparse distribution occurs. Finally, certainty of line is based on the votes number, so using too fine discretization or a parameter space of dimension higher than three may cause difficulties in the peak detection process. The same holds for clear but short segments that may not be detected.

For the first problem, *the probabilistic Hough transform* (PHT) by Kiryati et al. [41] and the *randomized Hough transform* (RHT) by Xu et al. [76] have been proposed. The *extended RHT* by Kalviainen et al. [38] realized a more efficient detection by applying RHT in the sub-windows randomly selected in the image.

For the second problem, quantization errors have been analyzed and discussed in the *high precision Hough transform* by Morimoto et al. [55] and in the efficient sampling methods by Goto et al. [24] and Van Veen et al. [74].

For the third short segment extraction problem, Princen et al. [65] proposed the *hierarchical Hough transform,* which uses hierarchical grouping process in conjunction with a local Hough transform.

The various HT variants were proposed in order to minimize the above mentioned computation and memory requirements, the major drawbacks of the standard HT. Some of them are based on a *coarse-to-fine* iterative search algorithm, e.g., the *adaptive Hough transform* (AHT) [33], the *fast Hough transform* (FHT) [46] and the *multiresolution Hough transform* (MHT) [3]. A set of reduced-resolution images is generated from the original image. Initially, the Hough transform is applied to the smallest image, hence using a very small accumulator array. Successive iterations use images and accumulator arrays of increasing sizes. The *AHT* uses a small accumulator array and the idea of a flexible iterative accumulation and search strategy for peak detection. It first analyzes the accumulator array at low resolution and then continues down into the neighbouring area of the peak at subsequent iterations. The starting binary edge image and the same accumulator array are repetitively used during all the iterations. The *FHT* is based on a hierarchical approach. The parameter space is gradually divided into hypercubes from low to fine resolution and performs the Hough transform only on the hypercubes with votes exceeding a selected threshold.

Further improvements of HT were introduced by Chatzis et al., namely the *fuzzy cell Hough transform* (FCHT) [9] and a combination of RHT and FCHT - *randomized* *fuzzy cell Hough transform* (RFCHT) [10]. The latter splits the parameter space into fuzzy cells with overlapped intervals of confidence. Murakami and Naruse [56] presented a 'partial to global' approach applying HT in small windows and extending the line to another area if needed; Kamat and Nagesan [39] used the butterfly-shaped spread of votes in the accumulator to adaptively define windows of interest around a detected peak of multiple line segments. Song et al. [71] came up with an idea of a boundary recorder to eliminate redundant analysis steps. They employed an image-analysis-based line verification method to overcome the difficulty of using a threshold to distinguish short lines from noise.

3.2.1 Hough Transform for Line Detection

In this section the simplest case of Hough transform - the *Hough linear transform* - will be presented. In general, using HT to detect lines consists of three steps: *accumulation*, *peak detection* and *line verification*. A pre-processing phase is usually necessary to extract feature points (*medial* or *edge* points) from the image.

At first, let us examine the mathematical background of the method in brief. Let (x_1, y_1) be a point in 2-D image space. Then a straight line passing through this point can be represented in the so-called *slope-intercept form* as $y_1 = mx_1 + b$, where m is the *slope* of the line and parameter b is its *intercept*. Hence, the straight line $y_1 = mx_1 + b$ can be represented as a point (b, m) in the slope-intercept space (also called *parameter space*). All of the infinitely many lines passing through (x_1, y_1) have to satisfy the equation for varying values of m and b. Rewriting this equation as $b = -mx_1 + y_1$ and considering the mb-plane, we get the equation of a single line for a fixed pair (x_1, y_1).

Now consider a second point (x_2, y_2). It has a line in parameter space associated with it as well. We will name the interesection of the two lines by (m', b'), where m' is the slope and b' is the intercept of the line $\overleftrightarrow{w}$ containing both points (x_1, y_1) and (x_2, y_2) in the original xy-plane (see Figure 3.1). As a matter of fact, all of the lines in parameter space, which are associated with the points contained on the line $\overleftrightarrow{w}$, intersect at the point (m', b').

Disadvantage of the *slope-intercept* representation lies in its instability caused by unboundedness of the two parameters. As lines get more and more vertical, the magnitudes of m and b grow towards infinity. Moreover, discretization of m is non-linear.

Duda and Hart [17] came with a solution based on different, *normal parametrization*. Two other parameters are used, commonly named ρ and θ. The parameter ρ represents the algebraic distance between the line and the origin, while θ is the angle of the vector from the origin to this closest point. Using this parametrization, the equation of the line can be written as

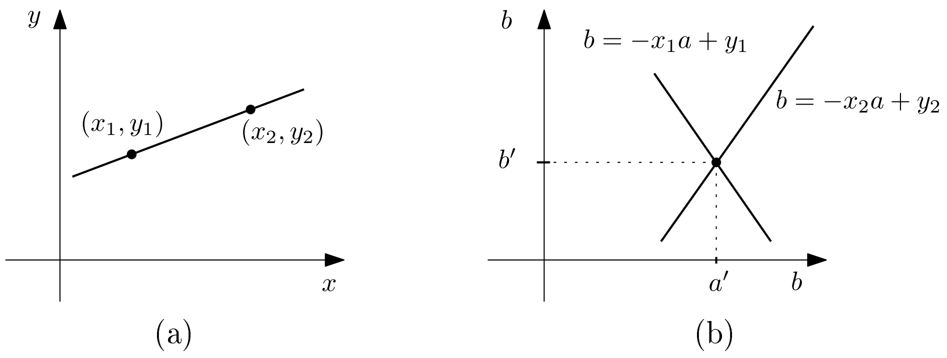


Figure 3.1: Hough tranform: (a) line in image space, (b) slope-intercept parametrization of the line.

$$y = (-\frac{\cos\theta}{\sin\theta})x + (\frac{\rho}{\sin\theta})$$

which can be rearranged to

$$\rho = x\cos\theta + y\sin\theta \tag{3.1}$$

To each line of the image it is possible to associate a couple (ρ, θ), which is *unique,* if $\theta \in [0, \pi)$ and $\rho \in \mathbb{R}$, or if $\theta \in [0, 2\pi)$ and $\rho \geq 0$. The $\rho\theta$-plane is sometimes referred to as *Hough space* (see Figure 3.2). This representation makes the Hough transform conceptually very close to the two-dimensional *Radon transform* [15].

Again, Hough transform maps all the pixels on a line into one point in the $\rho\theta$-plane. Instead of a a straight line, the trace is now a *sinusoidal* curve, as all the lines going through a point (x, y) obey the equation $\rho(\theta) = x\cos\theta + y\sin\theta$. Like before, N colinear points in the xy-plane lying on a line $\rho_i = x\cos\theta_j + y\sin\theta_j$yield N sinusoidal curves intersecting at (ρ_i, θ_j) in the parameter plane.

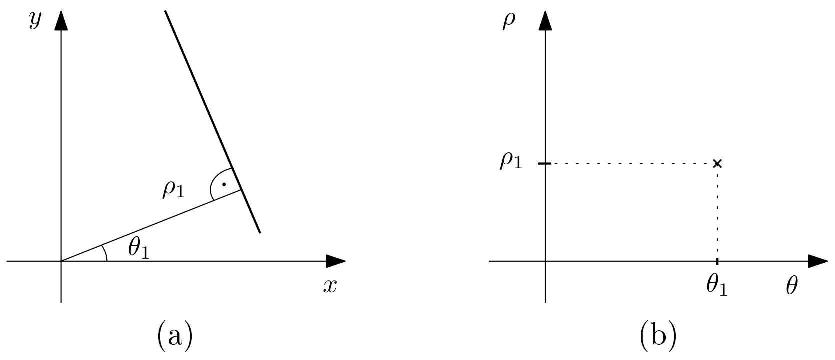


Figure 3.2: Hough tranform: (a) line in image space, (b) normal parametrization of line.

Thus, we managed to convert the problem of detecting colinear points in an image to the problem of finding concurrent curves in its Hough space. The *accumulation* and *peak detection* phases are described in more detail in Section 3.2.1.1.

The *line verification* step is supposed to find the exact location of line segments along the line. The basic method is to sequentially check the connectivity of feature points within the narrow strip area determined by the peak parameter (ρ_i, θ_i), the quantization interval $\Delta\rho$, and the sampling interval $\Delta\theta$. The feature points are searched for iteratively and the line equation is calculated frequently. Therefore, for large-sized images containing numerous lines this step may be more time-consuming than the previous two steps.

3.2.1.1 Implementation

For the computational purposes, the values of m and b are discretized by uniform subdivision of the parameter space into so-called *accumulator cells.* The dimension of the accumulator is equal to the number of unknown parameters of Hough transform problem. The cell at coordinates (i, j) has the accumulator value $Acc(i, j)$, initially set to zero. For each point (x_k, y_k) in the xy-plane, we iterate parameter m through all its allowed subdivision values, compute the other parameter b using the equation $b = -mx_k + y_k$ and round it to its nearest allowed value. For every pair (m_s, b_t), we increment the value $Acc(s, t)$ by 1. Consequently, at the very end of this algorithm, the value $Acc(u, v)$ is equal to the number of all points in the xy-plane lying on the line $y_u = mx_v + b$. The accuracy depends on the number of subdivisions of the parameter space. For an example of a Hough parameter space, see Figure 3.3.

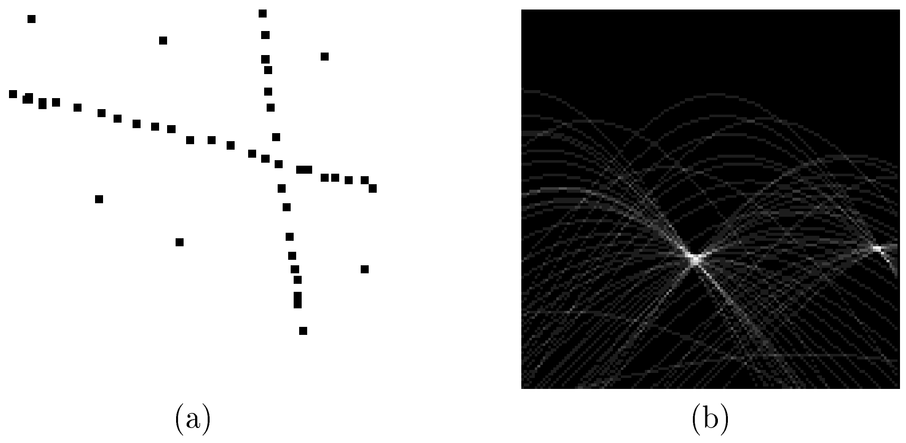


Figure 3.3: Hough transform - line detection: (a) original image, (b) parameter space with sinusoidal curves associated with the points in the image plane. Notice two bright peaks corresponding to the two lineal groups of points in the original image.

Detecting the local maxima in the accumulator is sometimes a non-trivial part of the problem. The simplest way of finding these peaks is by applying some form of threshold, or by finding the local maxima within an NxN neighborhood. The choice of N is important: using too large N will suppress some real lines, setting N too small will yield overlapping lines. Princen et al. [65] proposed an iterative global peak detection method. This method is more robust than clearing a rigid-size neighborhood, but it is very time consuming for a large-sized image due to the iterative accumulations.

Since the lines returned do not contain any length information, it is often necessary to find, which parts of the image match up with found lines.

3.2.2 Hough Transform for Curve Detection

It is straightforward to generalize the HT for detection of more complex curves that can be described by an analytic equation. Consider an arbitrary curve represented by an equation $f(x, a) = 0$, where a is the vector of the curve parameters. See Algorithm 3 for a general algorithm for Hough transform curve detection, as presented in [52].

Algorithm 3 Curve detection using the Hough transform

```
1: Quantize the parameter space within the limits of parameter $a$. The dimensionality
   of the parameter space is equal to the number of parameters of the vector $a$.
2: Create an $n$- dimensional accumulator array $Acc(a)$ with structure matching the
   quantization of parameter space; set all cells to zero.
3: For every foreground point $(x, y)$, increment all accumulator cells $Acc(a)$ by $\Delta Acc$
   if $f(x, a) = 0$ for all $a$ inside the limits used in step 1.
4: Local maxima in the accumulator array $Acc(a)$ correspond to realizations of curves
   $f(x, a)$ that are persent in the original image.
```

For instance, a circle can be parameterized as

$$(x - a)^2 + (y - b)^2 = r^2, \tag{3.2}$$

where the circle has center (a, b) and radius r. For each pixel x, all accumulator cells coresponding to potential circle centers (a, b) are incremented. The cell $Acc(a, b, r)$ is incremented in the case point (a, b) is at distance r from point x, and this condition is valid for all triplets (a, b, r) satisfying equation 3.2.

Because of the three curve parameters, the accumulator has to be three-dimensional and therefore circles require more computation and storage space to find than lines. Since the computation grows exponentially with the number of parameters, the Hough transform is typically used only for simpler curves, such as straight lines or parabolas.

3.2.3 Generalized Hough Transform

The generalized Hough transform is used when an analytical description of the feature we are searching for is not possible. Instead of a parametric equation describing the feature, we use some sort of lookup table, correlating locations and orientations of potential features in the original image to some set of parameters in the Hough transform space [4].

Still, even the generalized Hough Transform requires the complete specification of the exact shape of the target object to achieve precise segmentation. It allows detection of objects with complex, but pre-determined shapes.

Chapter 4

Lúthien

4.1 Target Platform and Environment

The program is destined for the Intel x86 architecture and runs on the GNU Linux platform - primarily developed under (K)Ubuntu 7.04, a free, Debian derived Linux-based operating system. There should be no obstructions regarding other major *nix distributions, either.

Lúthien requires *CFITSIO*, *ImageMagick* and *RTS2* libraries to be present (C.2).

4.2 Programming Language

The whole program core is written in object-oriented C++ programming language. There were several reasons to do so – the possibility to integrate the code with the RTS2 library and to derive classes from it; CFITSIO is a C library; benefits of object oriented approach, such as easily extensible and readable code and speed of C++ compiled code.

Bash scripts were used for testing purposes.

4.3 Program Interface

At the moment, *Lúthien* is a command line application to be launched using console or a shell script. In general, the command looks as follows

```
$ luthien {-h|-e|-r} -i IN1:IN2 [-o OUT1[:OUT2]] [-m MASK] [-n NUM] [-v]
```

Some typical examples and further information can be found in the User's Manual, in Appendix A.

Constants important for flexible and fast adjustment of the detection methods are included in the configuration file '`config.cfg`'. For more information, see Section A.4.1.

One of the possible future improvements is a graphical interface for the program arguments and configuration file setup. At the moment, it does not seem to be necessary.

4.4 Line Detection Implementation

The problem specification and some special characteristics of the supplied astrographic images provide us several ways how to fasten up the detection process, while keeping it as simple as possible. This section discusses such features and describes solutions chosen to exploit them.

Information redundancy

▷ Redundancy in time – Motion is a powerful cue to extract objects of interest from a background of irrelevant detail. For instance, in *tracking* of moving vehicles in a sequence of images, subtraction is used to remove all stationary components. What is left should be the moving elements, plus noise. The BOOTES system takes one widefield image per minute, in a continual series all night long. The difference between two consecutive images caused by the Earth rotation is relatively small. We can say that 'normal' sky objects (like stars, planets, sky background, as well as buildings on the image borders) are almost *stationary* components in the image. And then, meteorites (as short-term changes in the sky) represent moving objects. Thus, one of the basic ideas for *Lúthien* implementation is to always take a pair of two successive images and compute their pixel-by-pixel *difference* image. In order to bring out more detail, we can also perform a contrast stretching transformation. Such difference image has small brightness values. Difference image computation can be found in the pre-processing stages of all three *Lúthien* detection methods.

▷ Redundancy in spatial information – Typically, the FITS images processed by *Lúthien* have spatial resolution of around 4100 × 4100 pixels. Size of such FITS images is around 32 MBytes (or 20 MBytes as gzipped files). Therefore, the computational and memory costs are high. But thanks to high spatial information redundancy we do not need to work with all the pixels. The least computationally demanding method to extract only a representative part of the image pixels is directly supported by the CFITSIO library. It offers a special FITS file opening mode in which only every n-th pixel in each axis direction is loaded. Class `L_Rts2Image` uses this mode in the `loadFitsPixelData` method for the same pupose. For instance, for $n = 2$ we get only one fourth of the image data, which means substantial acceleration of the calculation usually without deterioration of results accuracy.

▷ Border areas – In widefield camera images, valuable information forms a 'circle' in the central part of the image. The rest in the border areas (especially in corners) is not only useless, but even undesirable, because it can significantly spoil the line detection results (like parts of the telescope or buildings with lineal features). An easy solution is to use a mask image to eliminate these areas. It is optional, but a highly recommended step.

▷ Redundancy in brightness value resolution – The FITS images generated by the BOOTES system (see 5.3) processed by *Lúthien* are 16-bit unsigned integers, thus having value range of 0 - 65535. Usual 2-D greyscale images have only 256 levels, but astrophotographic images are very often of low contrast, using only as little as one tenth or one twentieth of the brightness scale. Using histogram equalization we get a high dynamic resolution image. Distribution of pixel values in this wide range also simplifies the segmentation.

Meteorite characteristics

▷ High brightness – Meteorites and stars are much brighter than the residual sky background. This helps us immensely to segmentationally separate them as foreground objects of interest. Even simple thresholding works in this case. Preprocessing stages of the detection methods take advantage of this fact.

▷ Meteorite geometry – A meteorite (or its trail, to be more accurate) is basically a bright line on dark sky background. Usually, there are no other lineal structures, apart from man-made structures on the borders (and those can be eliminated using a mask). Thus, the Standard Hough transform for lines (3.2.1) can be used. Thinning does not change the overall shape of a line. A meteorite has also some typical minimum thickness (erosion can help eliminate noise and small stars), length (satellite tracks filtration), and high elongatedness (in comparison to, e.g., stars). During the detecion process, all of this additional knowledge is used.

In following sections, the detection methods are presented in more detail. See also Section A.4 for further information.

4.4.1 Hough Transform Method

Probably the most accurate and robust of the three detection methods is the one based on the Hough tranform. The input data is loaded, histogram equalization and difference image computation are executed. At this moment, there are two possible pipelines to process data for the core Hough transform.

The first one computes the Sobel edge detection 2.2.3.2, thresholds and dilates its output to get a 'sobel mask image'. The original difference data is then thresholded and masked with this sobel mask. Masking with a regular mask image is then (optionally) applied. On the other hand, the second pipeline thresholds the difference data and applies morphological operation *closing* (dilation followed by erosion, see 2.3) *directly* on the binary result. The idea is to eliminate noise (like one pixel elements, the detected line is thick enough not to dissappear). The dilation and erosion steps can be repeated more times (see A.4.1). The histogram of difference data is not ideally bipolar, but the brightness difference between the foreground (meteorites, stars) and background (dark sky) is usually sufficient for choosing a good threshold value – then the second method works well enough. But sometimes one of the input images is strongly affected (for instance, by reflectors of a car), causing too big difference in the input images. In such cases, the first method performs much better, as it works with edge pixels instead of image pixels.

The binary images prepared by the pre-processing and morphology stages are thinned and passed to the core Standard Hough transform. It iterates through all points (black points in the binary image in this case), computes the normal parametrization values and accumulates them in so-called Hough space (3.2.1). One way to improve its speed is to reduce number of processed points. Therefore, thinning creating a skeleton is employed (2.3). Peaks are detected using the local maxima or (by default) butterfly peak detection method described in [39].

Consequently, the detected lines are verified. One reason for the time-inefficiency of common HT line verification methods is that the accumulator itself does not provide any information about the ending points of the line, only in which strip area the feature points lie. Hence, every position within the strip area has to be checked, or all feature points with the known θ are recalculated to get their ρ. Because in most cases only a small part of the strip contains the feature points, neither way is fast for a high-resolution image. The idea (presented in [71]) is to add a *boundary recorder* to each parameter cell to record the minimum range containing the feature points corresponding to the parameter. The boundary is actually represented by two feature points, called *low boundary* and *up boundary*. According to Equation 3.1, one dimension of the image coordinates can be calculated from another dimension. Thus, to save some memory, we only record one (x- or y-) dimension of the coordinates. The choice depends on slope – in case $45° < \rho < 135°$, the line is nearly horizontal, so the x-dimension coordinate is chosen to record the boundary. Otherwise, the y-dimension coordinate is chosen. The initialization and recording algorithm in pseudocode is shown in Algorithm 4.

Algorithm 4 Boundary recorder

```
1: Init:
   Accumulator[theta][rho] = 0;
   LowBoundary[theta][rho] = max(xMax, yMax);
   UpBoundary[theta][rho]  = max(xMin, yMin);
2: Recording: (point (x,y) contributing to parameter (theta, rho))
   Accumulator[theta][rho] += 1;
   if (45° < rho < 135°) {
     LowBoundary[theta][rho] = min(LowBoundary[theta][rho], point.x);
     UpBoundary[theta][rho]  = max(UpwBoundary[theta][rho], point.x);
   } else {
     LowBoundary[theta][rho] = min(LowBoundary[theta][rho], point.y);
     UpBoundary[theta][rho]  = max(UpwBoundary[theta][rho], point.y);
   }
```

Two thresholds are used; *minLength* for the minimum acceptable line length and *maxGap* for the maximum acceptable gap length. After the peak detection, all peaks higher than *minLength* are taken and stored in a list in descending order of peak value (number of votes). The line verification begins from the head of the peak list. So, the more important lines are processed sooner, the global peak being the first one. Using the boundary recorder, we only need to analyze a small part of the original image determined by the peak coordinates. The pixels of the verified line segments are then removed from the image, by setting their value from black to white. The boundary points coordinates are calculated and the image pixels along the straightline connecting these boundary points inspected. For that purpose the Bresenham algorithm for straight line [77] is used. The algorithm detects all segments, which are longer than *minLength* and do not contain gaps longer than *maxGap*.

Finally, the line filtering stage finds similar lines (in slope and intercept), throws out duplicates and connects parts of the same line.

4.4.2 Edge Detection Method

The second detection method is based on the Canny edge detector technique. Again, after the input data is loaded, histogram equalization and difference image computation is executed. Masking with a mask image is then (optionally) applied.

Edge detection using the Canny edge detector is performed (see Section 2.2.3.3). The image is smoothed with a Gaussian mask to eliminate the noise. The larger the width of the Gaussian mask, the lower the detector's sensitivity to noise. The edge strength is found by taking the gradient of the image. The Sobel operator is used for computation of gradient approximation. It uses a pair of 3×3 convolution masks; one for the gradient in the x-direction and the other for the gradient in the y-direction. The magnitude of the gradient (edge strength) is then approximated using the Equation 2.4. Non-maximal suppression eliminates unimportant edge pixels. In the hysteresis step, the initial segments of strong edges are found using the high threshold, while the low threshold is used for edgel linking.

The data is then converted to binary representation, dilated and eroded using a 3×3 mask. Consequent blob detection stage searches for bright line-blobs of pixels using simple fill algorithm. Only blobs big enough, long enough, and with sufficient elongatedness are accepted. Parameters of the lines corresponding to the conforming blobs are computed.

In the end, the line filtering stage finds similar lines (in slope and intercept), throws out duplicates and connects parts of the same line.

4.4.3 Region Growing Method

The final detection technique is based on simple region growing. The input data is loaded, histogram equalization and difference image computation is executed. Masking with a mask image is then (optionally) applied and the data is thresholded.

The data is then converted to binary representation, dilated and eroded using a 3×3 mask to connect segments of the foreground structures and to close holes in them.

Blobs of bright star pixels are detected using simple region growing method. The thresholded data gives us the seed pixels position. A neighbouring pixel is marked as a part of the blob, if its brightness value is higher than a certain value depending on the seed pixel value. Moreover, only line-blobs satisfying further conditions are accepted – minimum pixel size, minimum Euclidean length and minimal elongatedness. Parameters of the lines corresponding to the conforming blobs are computed.

Again, the line filtering stage finds similar lines (in slope and intercept), throws out duplicates and connects parts of the same line.

Chapter 5

Software and Hardware Environment

5.1 FITS File Format

In the following paragraphs some basic facts about FITS file format are presented, as most of the astronomical data, as well as all input data for the *Lúthien* program, are stored using just this file format.

Flexible Image Transport System (abbrev. *FITS*) is a data format designed to provide a way to conveniently exchange astronomical data (not digital imagery only) between systems with differrent standard internal formats and hardware. It has become the standard file format used to store astronomical data files. A FITS data file contains a sequence of *Header Data Units* (HDUs) of two basic types - *images* and *tables*. The header consists of *'keyword=value'* statements describing the organization of the data in the HDU and the format of its contents. It may provide additional information, e.g., about object being photographed, observer, exposure time, instrument status, calibration, or the history of the data. The data itself follows, structured as the header specified. A single FITS file may contain multiple images or tables. The first HDU in a FITS file has to be an image (but it may have zero axes), called *primary array*. Any additional HDUs in the file (which are also referred to as *extensions*) may contain either an image or a table.

There are two subtypes of FITS tables: *ASCII* and *binary*. ASCII tables store the data values in an ASCII representation, whereas binary tables store the data values in a more efficient binary format. Binary tables are generally more compact and support more features (wider range of datatypes, vector columns, etc.) than ASCII tables.

FITS images typically contain a 2-dimensional array of pixels representing an image of a part of the sky. But they can also contain 1-D arrays (a spectrum or light curve), or 3-D arrays (a data cube), or even arrays of data of higher dimension. An image may also have zero dimensions. In that case it is referred to as a *null* or *empty array*. The supported datatypes for the image arrays are 8-, 16-, and 32-bit integers, and 32- and 64-bit floating point real numbers. Both signed and unsigned integers are supported.

The FITS images generated by the BOOTES system (see 5.3) processed by *Lúthien* are 16-bit unsigned integers, with value range of 0 - 65535. Usual 2-D greyscale images have only 256 levels, but astrophotographic images are very often of low contrast, using only as little as one tenth or one twentieth of the brightness scale. Coding into 256 levels would cause unacceptable loss of information caused by quantization. Space resolution of the tested input images was 4090 $\times$ 4090 pixels and 4098 $\times$ 4098 pixels. Size of such FITS images is around 32 MBytes (or 20 MBytes as gzipped files).

For further details about the FITS file format, visit [73], [20] or [70].

5.1.1 FITS Software

FITS file format is not very well known by ordinary users, as it is used mostly for scientific purposes; software not designed for astronomy generally does not support and accept FITS files, not even for viewing.

Software for manipulating FITS files can be divided into three main groups - *libraries* providing interface for various programming languages, *editors* and *browsers / viewers*. Their most important representatives are shortly introduced in this section.

5.1.1.1 CFITSIO - A FITS File Subroutine Library

CFITSIO is a library of C and Fortran subroutines intended for reading and writing data files in FITS data format, providing high-level routines for reading and writing FITS files that insulate the programmer from the internal complexities of the format. CFITSIO also provides many advanced features for manipulating and filtering the information in FITS files. For documentation and other information, visit [20].

There are interfaces for calling CFITSIO from many programming languages available - *C++* (*CCfits*), *C#* and the *.NET Platform* (FitsLib), *Perl* (CFITSIO.pm), *Tcl* (fitsTcl), *Python* (pCFITSIO), *Ruby* (RFits), *S-lang* (CFITSIO wrappers for the S-lang programmer's library and interpreter) and *MatLab* (MFITSIO).

CCfits is an object oriented interface to the CFITSIO library for the C++ language. It is written in ANSI C++ and implemented using the C++ Standard Library with namespaces, exception handling, and member template functions. The programmer manipulates FITS objects simply by passing filenames and lists of strings that represent HDUs, keywords, image data and data columns. For additional information and download, visit [8].

5.1.1.2 FTOOLS

FTOOLS is a general collection of over 200 ANSI Fortran or ANSI C utility programs, Perl scripts and Tcl scripts designated to create, examine, or modify data files in the FITS format, including *fverify* (verifies a file conformance to the FITS Standard), *fcopy* (selectively copies parts of FITS file to a new file) and *ftlist* (prints the contents of FITS headers, images, and tables).

A graphical interface is available and users have the option of installing the entire package with many high energy astrophysics specific routines, or just a core set that contains only the routines to perform general operations on FITS files. FTOOLS can be built either as a set of stand-alone executable tasks or as a package within IRAF.

For more details and download source, visit [58].

5.1.1.3 FITSVERIFY - A FITS File Format Checker

The goal of the FITSVERIFY program is to verify the conformance to the standard document of any FITS format disk data file, checking keywords and data. FITSVERIFY is a stand-alone version of *fverify* distributed as a part of the *FTOOLS* software package. FITSVERIFY is still under development, thus it may be unstable under special circumstances.

The program is available as an executable binary file for Solaris, Linux and Windows platforms. The source code is also available for building on other platforms.

For further information, see [58].

5.1.1.4 Fv - An Interactive FITS File Editor

Fv is an easy-to-use general-purpose FITS file graphical editor able to manipulate virtually all aspects of a FITS file and perform basic data analysis of its contents. The *fv* software is small, completely self-contained or included as a standard part of the FTOOLS distribution. It can be used with the DS9 image display (see next Section).

Fv displays images with pan and zoom, manipulates the color table, edits FITS header keywords and data values, includes a summary window listing contents and size of all extensions, produces line plots of the values in two or more columns of a FITS table, with export to a PostScript file available. It is currently supported on many Unix platforms; Windows and Macintosh versions are under development.

For more details and download source, see [58].

5.1.1.5 FITS Browsers & Viewers

SAOImage DS9 is an astronomical imaging and data visualization application. Currently it supports advanced features such as multiple frame buffers, mosaic images, tiling, blinking, geometric markers, colormap manipulation, scaling, arbitrary zoom, rotation, pan and a variety of coordinate systems, FTP and HTTP access. DS9 provides tools for easy communication with external analysis tasks and is highly configurable and extensible.

For more details about DS9 and download source, visit [2].

Other FITS browsers are the *ESA/ESO/NASA Photoshop FITS Liberator* for Adobe Photoshop, *ADC FITS Table Browser* and *NCSA FITS Browser*, for viewing purposes *NRAO FITS viewers* (*FITSview*, *MacFITSView*, *XFITSview*), *MSI Windows FITS viewer*, *SAO R&D Software Suite* or *netPBM* can be used.

5.2 RTS2 Package

Lúthien takes advantage of `Rts2Image` class, as class `L_Rts2Image` is its successor. The `Rts2Image` class is a part of the RTS2 library package, originally created by Mgr. Petr Kubánek (the advisor of this work) as his diploma thesis. In this section we will provide some more information about this package.

Remote Telescope System, 2nd Version (abbrev. *RTS2*) is an integrated software package designed for full robotic automatization of astronomical observations. The system takes care of the whole image acquisition process, beginning with target selection from a database, ending with processing of acquired images while allowing certain level of user control (ranging from simple on-off-standby regimes to more advanced scripting). At the moment, five telescopes on three continents are controlled by the system, e.g., the BOOTES system (see Section 5.3). At the beginning, the RTS2 was intended to control telescopes dedicated to observe optical opposites of gamma-bursts. Gradually, it has become a much more general system for controlling robotic telescopes.

One of the biggest advantages of the program is the full *remote control* via SSH protocol. Various commands like setting enviroment variables, system upgrades, viewing system logs and image manipulation (to name a few) can be executed using virtually any computer with internet connection.

The RTS2 runs on Linux platform - primarily developed for Debian, but installed on RedHat and SuSE distributions as well, without any apparent problems. No serious difficulties when getting it to work for other major *nix distributions are expected.

Other important features of RTS2 are, e.g., integrated on-line astrometry using either *astrometry.net* package or any other command-line astrometry package, target creating from SIMBAD (fixed position target), gamma-ray bursts alerts processing module, script-driven GRB observation, usage of PostgreSQL database to store observation details, *ncurses* based monitoring (`rts2-mon`), priority-function (selector), planned (night schedule), manual (from `rts2-mon`) and urgent (GRBs) observations. The package is still under development and the authors plan to add many new features in the near future.

For more information, see [42, 18, 67].

5.3 Bootes

According to [29], there are at least four major fields of recent and future activities of the *High Energy Astrophysics Group at the Astronomical Institute of the Academy of Sciences of the Czech Republic* in the wide–field sky imaging and sky monitoring. First, optical monitoring and analyses of selected targets with robotic telescopes – BART, BOOTES, SUPER–BART, BOOTES–IR. Analyses of archival sky patrol plates come second, the third goal being the satellite experiments: INTEGRAL OMC, LOBSTER and finally, CCD sky monitoring as long term analyses.

The valuable and unique scientific information recorded by the telescope systems BART and BOOTES is not yet fully taken advantage of. By developing new analysis software, such as *Lúthien*, we are trying to get closer to this goal.

Lúthien was originally created to process astrographic images created by *BOOTES*, a robotic telescope system used for optical monitoring and analyses of selected targets. It is a result of Spanish–Czech (IAA-CSIC, Granada - AsÚ AV ČR Ondřejov) cooperation, located in Spain (one at the Observatory de Sierra Nevada in the Sierra Nevada mountains, approx. 2890 m above the sea level). The fully automated telescopes use various CCDs. At the moment, the most advanced telescope is BOOTES-IR (60 cm aperture, near Veleta peak, Sierra Nevada range).

The BART and BOOTES systems have been developed for a number of purposes - automatic GRB follow–up observations, monitoring and photometry of selected triggers (mostly blazars, QSOs, AGNs and selected binary galactic sources) and testing of newly developed software (with astronomy and computer science students involved).

For further information about BART and BOOTES systems, see [18, 29].

Chapter 6

Testing Results

This chapter summarizes experimental testing carried out in order to compare the detection alorithms implemented in the *Lúthien* software package. The results are interpreted according to the initial problem specification (see Section 1.2).

Testing Environment

- ▷ *Hardware configuration:* notebook PC, 2-GHz Intel Pentium Core Duo processor, 2 GB of RAM.
- ▷ *Software configuration:* Kubuntu 7.04 x86, kernel version 2.6.20, RTS2 library – ver. 0_5_0, libnova-0.12.1, CFITSIO library 3.040 and ImageMagick library of version 6.3.5.
- ▷ *Tested images:* `07_20070728034400.fits`, `07_20070728034600.fits`, `10_20060606023900.fits`, `10_20060606024000.fits` and complete image sequence
`20061118193419.fits` – `20061118231730.fits` (70 images in total).

The testing script consisted of following commands (one for each method):

```
$ ./luthien {-t 1|-t 2|-e|-r} -n 2 -i IN1.fits:IN2.fits
                                              -m mask6_vector.png
```

The software was tested using real input images taken by the BOOTES system in Spain, Sierra Nevada. The results were measured under the same circumstances and configuration settings. The input images were analyzed in pairs, in sequence identical to the one they were taken in by a camera (all the pictures were taken by single camera).

However, some of them were affected by wrong camera configuration (dark images of very low contrast), in few cases substantial difference between two images in a pair was apparent. This difference was caused by special conditions, for instance, a car passing by (one image brightened by car reflectors); one image has been even taken while the roof of the observatory was still closed. The deteriorating impact of this divergence made the detection often difficult, mostly resulting in a number of false positives, or assignment of a detected line to the other image of the pair (at correct location). But in many cases, the detection didn't fail inspite of the obstructions.

Thus, in the results summary two cases for each detection method are presented – one for error-free images and one taking all the images into consideration. The results measured for the implemented detection methods are presented in Table 6.1. *'ED'* stands for *edge detection* method, *'HT1'* and *'HT2'* are the two *Hough transform* pipelines and *'RG'* denotes the *region growing* method.

	Meteorites (8 / 13*)		Images (7 / 11*)		Runs (6 / 8*)	
	found	%	found	%	found	%
ED	6 / 8*	75 / 61.5*	6 / 8*	86 / 73*	5 / 6*	83 / 75*
HT1	6 / 7*	75 / 54*	6 / 7*	86 / 64*	5 / 6*	83 / 75*
HT2	6 / 8*	75 / 61.5*	6 / 8*	86 / 73*	5 / 6*	83 / 75*
RG	6 / 9*	75 / 69*	6 / 8*	86 / 73*	5 / 6*	83 / 75*

Table 6.1: Detection results. Asterisk * denotes results with corrupted images taken into consideration.

All of the presented detection methods seem to return very similar results, in almost all cases finding the same meteorite trails. The first column presents the detection efficiency – number of meteorites found manually to number of trails detected by the program ratio. The second column brings the same information about the number of images containing meteorites; the third column analyzes whole runs (*'run'* meaning one program analysis of a pair of images). We can see that all methods achieved accuracy of around 75% in detecting meteorites and are able to find approximately 85% of all interesting images (those containing at least one meteorite). Assuming that all the 'marked' images will be re-checked by a human operator, the latter percentage is of more importance.

Failure to detect some of the meteorites has various reasons. Already mentioned 'difference corruption' causes problems especially for the region growing method, as false seed points are created. The principle step of Hough transform is peak detection in the parameter space. Thus, weaker lines can sometimes be missed because of a much stronger response to another line (again, this is not a real problem, since the image is marked as interesting anyway). In other cases it takes more time to find the right program settings.

Besides finding the correct images with meteorite trails, low count of found false positives is fundamental. Table 6.2 tell us that the ratio of false alarms number to all runs number is as low as 3% or even less, when using error-free images. Even under special circumstances, edge detection and Hough transform type 1 work best, having almost no false positives. On the other hand, region growing and the second type of Hough tranform generate a false detection every tenth run. But this number falls to zero drastically as soon as we take into account only non-corrupted pictures.

	False positives in total			
	total images	total runs	false alarm runs	% of all runs
ED	0 / 0*	0 / 0*	0 / 0*	0 / 0*
HT1	0 / 1*	0 / 1*	0 / 1*	0 / 2.9*
HT2	1 / 9*	1 / 6*	1 / 4*	2.9 / 11.4*
RG	0 / 7*	0 / 6*	0 / 3*	0 / 8.5*

Table 6.2: False positives detection rate comparison. Asterisk * denotes results with corrupted images taken into consideration.

Speed of the program was one of the most important specification requirements. Table 6.3 shows information about time complexity of the tested methods. Obviously, all of them proved to be fast enough for real-world, practical use. The region growing method is statistically the fastest one (only 2.55 seconds in average), followed closely by Hough transform and more time-demanding edge detection. In general, none of the methods takes more than 10 to 12 seconds to run. Considering this fact, possibility of running more than one test for every pair seems to be realistic. Moreover, as we always analyze two images at a time, we actually can use *two* time-windows for our image processing, instead of one. And the more different methods used, the more accurate the results.

	Detection Time / Full Time (including loading data) [in sec.]		
	minimum	maximum	average
ED	9.35 / 11.61	9.72 / 12.07	9.50 / 11.81
HT1	4.33 / 6.44	6.64 / 8.82	4.91 / 7.06
HT2	3.39 / 5.66	11.62 / 13.92	6.56 / 8.87
RG	2.35 / 4.65	3.64 / 5.90	2.55 / 4.86

Table 6.3: Detection methods speed comparison.

The target hardware for *Lúthien* is a 2-GHz Intel Pentium Quadro-core desktop PC with 2 GB of RAM (and possibility of further upgrades), which makes it more efficient than the testing configuration. Thus, even higher speed of the program is expected.

The time complexity comparison with complete information about all 35 program runs si presented in Figure 6.1.

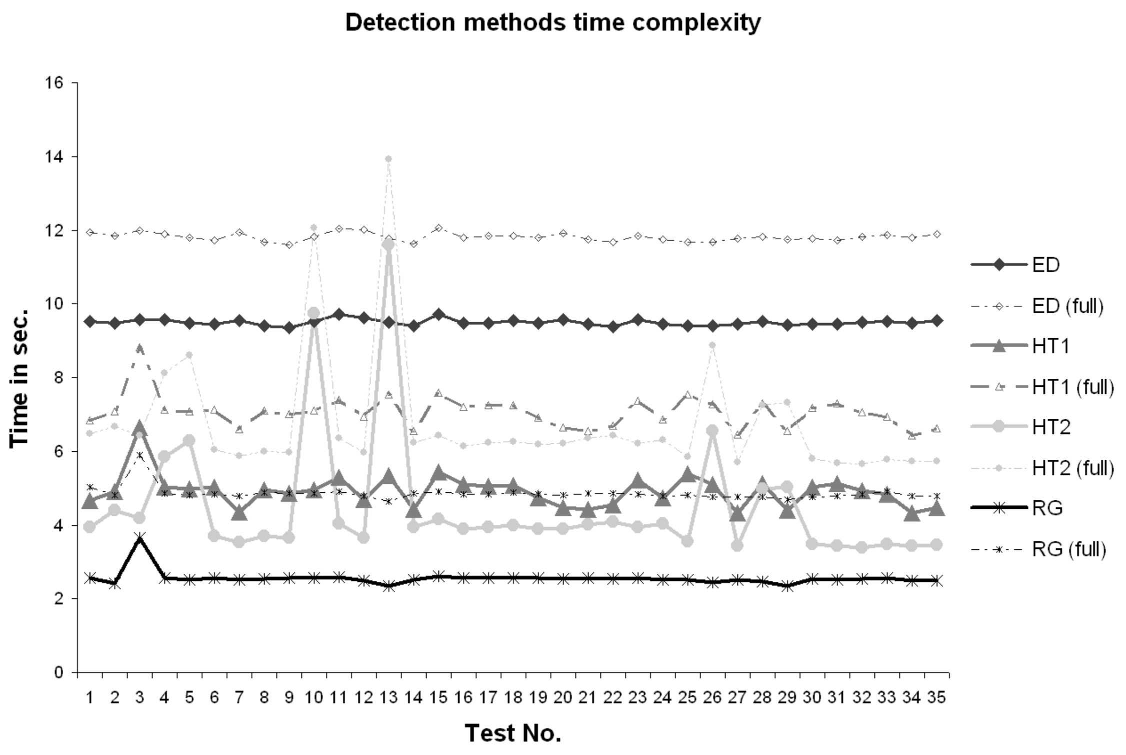


Figure 6.1: Time complexity comparison of the detection methods. For each method times of the detection process itself (e.g., *ED*) and times including the input data loading (e.g., *ED (full)*) are available.

All in all, none of the methods is an obvious winner. Region growing and type 2 Hough transform are very fast, but false positives can (in special cases) cause trouble. Slowlier edge detection and type 2 Hough transform seem to be more robust when dealing with corrupted images. Cooperation of more than one method could be a good solution. However, each of them is successful enough to be an alternative to the others. This possibility to choose from various image processing approaches is definitely an advantage of the *Lúthien* package.

6.1 Examples

In this section, three examples of typical input image processing with temporary results are presented. The original files are high-resolution and, as a matter of fact, only a small part of them is interesting. Therefore, for each step in the process only a cutout (at the same position) is shown. The full-sized files can be found on the enclosed CD-ROM in the `test_fits/ (input files)` and `output_fits/` (output files) directories. The output images are named `1_XY_NUM_NAME_.png`, where XY denotes the detection method used ('`HT`' for Hough transform, '`RG`' for region growing and '`ED`' for edge detection), `NUM` is its serial number and `NAME` gives further stage description.

Some of the images are inverted for the sake of better readability.

6.1.1 Hough transform method example

Input files: `20061118222705.fits` and `20061118222837.fits`. Command used:

```
$ ./luthien -t 1 -n 2 -i test_fits/20061118222705.fits
             :test_fits/20061118222837.fits
            -o output_fits/HT1_res1.png:output_fits/HT1_res2.png
            -m img/mask6_vector.png -v
```

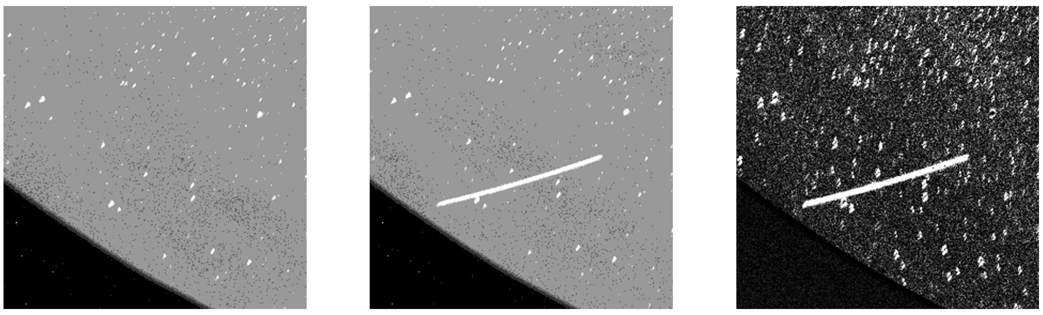
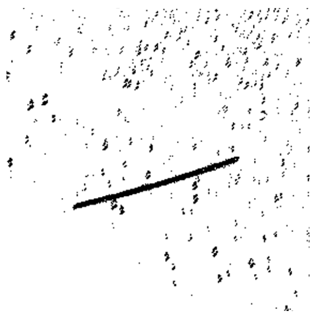
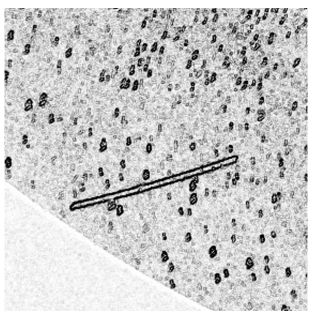

(a) (b) (c)

Figure 6.2: Hough transform method: (a) original image #1, (b) original image #2, (c) difference image.


(d)


(e)

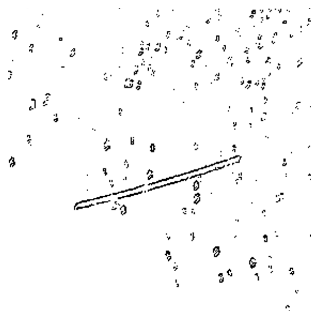
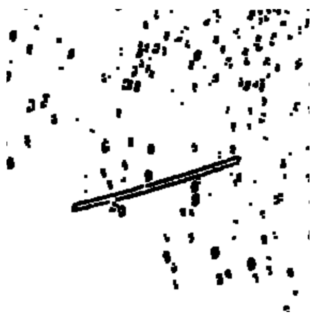
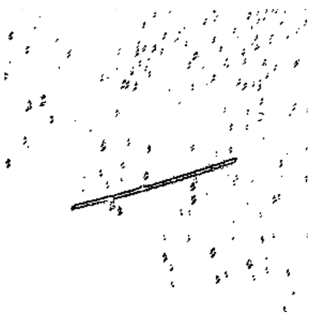

(f)

Figure 6.3: Hough transform method: (d) thresholded difference image, (e) Sobel edge detector applied on the original difference image (see (c)) , (f) thresholded Sobel image.


(g)


(h)

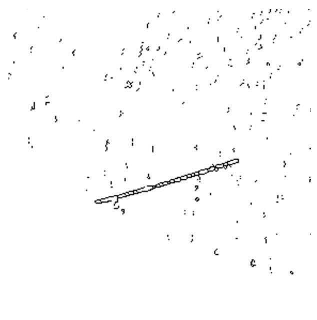
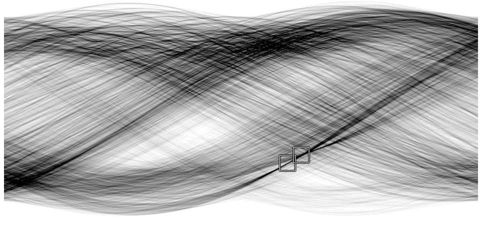

(i)

Figure 6.4: Hough transform method: (g) dilated Sobel image, (h) thresholded difference image (see (d)) masked out by dilated Sobel image (see (g)), (i) thinning result.


(a)

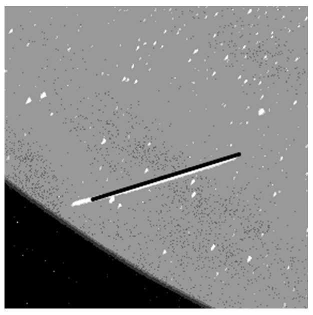

(b)

Figure 6.5: Hough transform method. (a) the accumulator array with peak marked, (b) the output image with a found line segment, coloured in black.

6.1.2 Edge detection method example

Input files: `10_20060606023900.fits` and `10_20060606024000.fits`. Command used:

```
$ ./luthien -e -n 2 -i test_fits/10_20060606023900.fits
             :test_fits/10_20060606024000.fits
            -o output_fits/ED_res1.png:output_fits/ED_res2.png
            -m img/mask6_vector.png -v
```

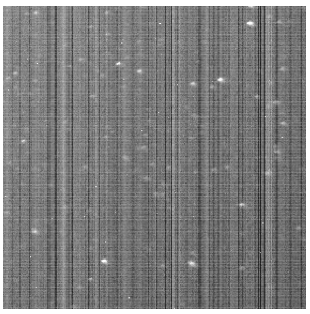
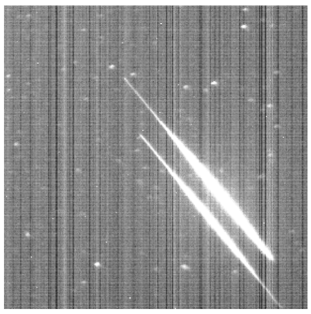
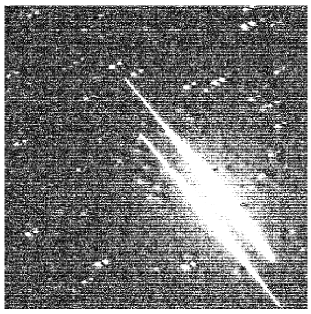

(a)

(b)

(c)

Figure 6.6: Edge detection method: (a) original image #1, (b) original image #2, (c) difference image.

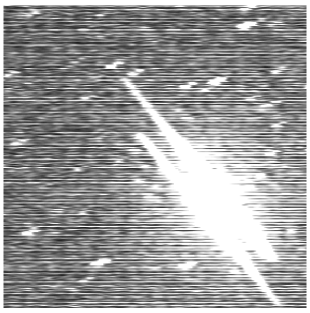
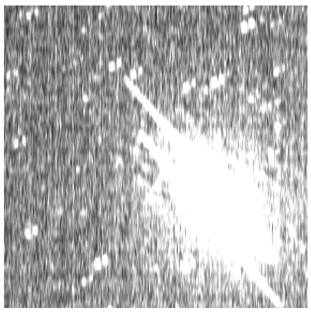
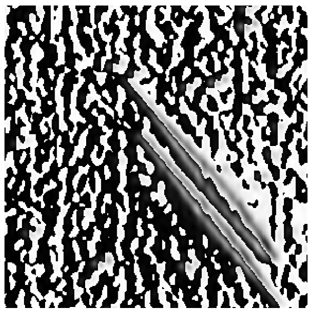
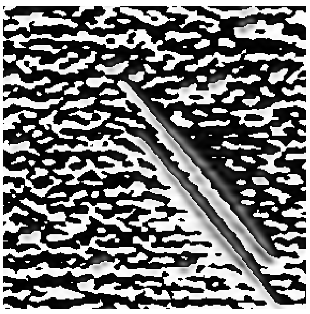
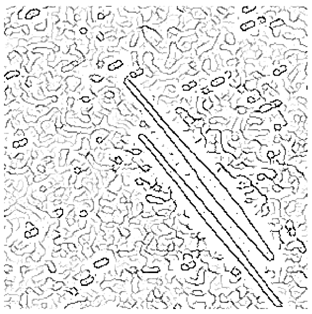

(d)

(e)

(f)

Figure 6.7: Edge detection method - Canny: (d) Gauss filtration in x-direction, (e) Gauss filtration in y-direction, (f) gradient magnitude in x-direction.


(g)


(h)

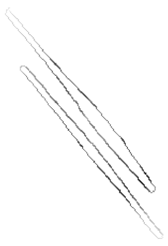
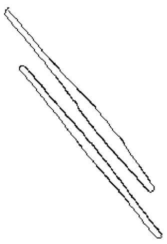
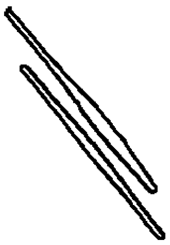

(i)

Figure 6.8: Edge detection method - Canny: (g) gradient magnitude in y-direction, (h) non-maximal suppression result, (i) thresholding with hysteresis applied.


(j)


(k)

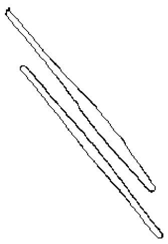

(l)

Figure 6.9: Edge detection method: (a) edge pixels thresholded, (b) morphological dilation, (c) morphological erosion.

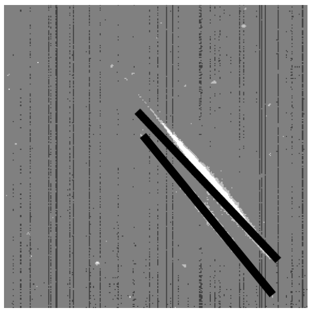


Figure 6.10: Edge detection method: the output image with found line segments, coloured in black.

6.1.3 Region growing method example

Input files: `07_20070728034400.fits` and `07_20070728034600.fits`. Command used:

```
$ ./luthien -r -n 2 -i test_fits/07_20070728034400.fits
               :test_fits/07_20070728034600.fits
              -o output_fits/RG_res1.png:output_fits/RG_res2.png
              -m img/mask6_vector.png -v
```

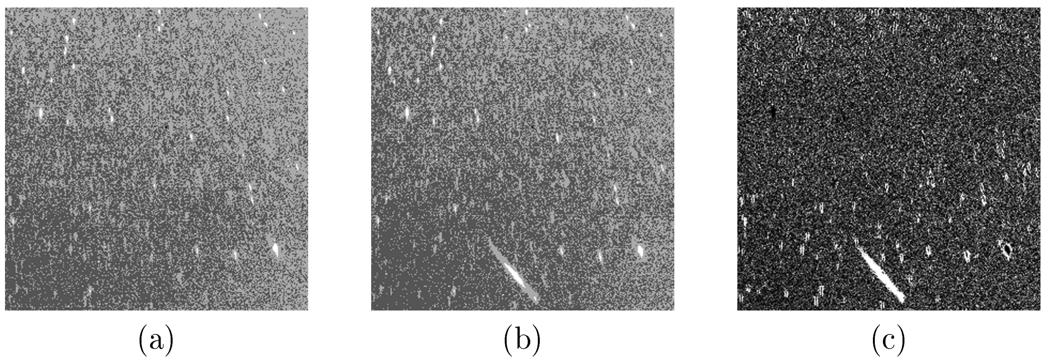


Figure 6.11: Region growing method: (a) original image #1, (b) original image #2, (c) difference image.

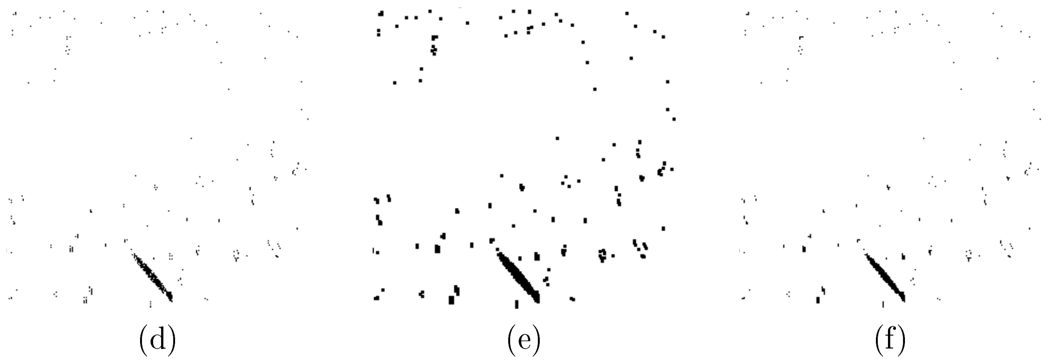


Figure 6.12: Region growing method:(d) threshold, (e) morphological dilation, (f) morphological erosion.

(g)

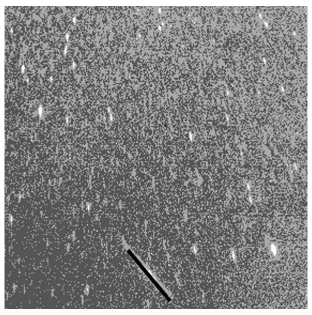

(h)

Figure 6.13: Region growing method:(g) blobs accepted in filtration stage, (h) the output image with a found line segment, coloured in black.

Chapter 7

Conclusion

The aim of this work was to create a meteorite detecting software tool able to process large-sized astrography FITS images created by a camera of BOOTES system and prove its applicability in real-world scenarios.

In the theoretical part of this thesis basic definitions and various image-processing techniques were presented, with more accent put on the approaches suitable for practical implementation. The reader was introduced into problematics of image pre-processing and morphological operations. Following part focused directly on line detection methods, especially the Hough transform and the Canny edge detection. Fundamental information about the FITS file format, the BOOTES telescope management system and the RTS2 library was discussed.

Description of the chosen algorithms is given in the practical part. Three different approaches were implemented – two based on (already mentioned) Hough transform and Canny detector, the third one uses a simple region growing segmentation. User's and developer's manual along with the generated documentation should provide a simple way to understanding, extending or patching the package source code. Experimental testing proved that (after some settings adjustment) all of the methods can be quite successful and their practical use is possible. Speed of the solutions allows multiple testing for each image pair.

The road to the end of the work has been long and thorny. Some decisions were made right, others were not and should have been reconsidered. When looking back, I naturally see a lot of things that could have been done otherwise, better. But that is the story of every human creation. On the other hand, this thesis gave me an excellent opportunity (and a good reason) to revise my long forgotten C language skills; I had to practically implement some of the algorithms I have only read about before. Writing such a large-scale text is definitely a good experience, as well.

I believe the program is extensible enough to make future feature addition possible and easy. For instance, I could think of a graphical user interface for program settings setup, adding some kind of line width detection algorithm, parallelization of the program run to utilize multi-processor computers, etc.. Ability to reliably distinguish between meteorites and trails made by planes or satellites is a big challenge, too.

Appendix A

Lúthien - User's Manual

Lúthien is an integrated software package designated to process high-resolution wide-field FITS images in order to detect meteorite trails. This appendix is a user's manual for version 1.0 of the package.

Author

Igor Gomboš

Feedback

Please direct any comments or suggestions about this document to:
luthien.project@gmail.com.

Acknowledgements

This documentation was created in LaTeX (http://www.latex-project.org/) using the LyX document processor (http://www.lyx.org/) .

Version

First edition. Published 14 December 2007.

A.1 What is Lúthien?

Lúthien is an integrated software package designated to process high-resolution wide-field FITS images in order to detect meteorite trails. It implements three different detection methods based on the Hough transform, simple region growing and Canny edge detector.

It was developed in C++ programming language and runs on Linux platform. The software uses fragments of the RTS2 library package, software package designed for full robotic automatization of astronomical observations, created by Mgr. Petr Kubánek (for further details, see Section 5.2).

The detector's output can be saved as images of various formats. For image output the ImageMagick library routines are employed.

A.2 Minimum Requirements

The RTS2 software package and ImageMagick library (see Section C.2) have to be installed on the system. The CFITSIO library for FITS files manipulation (see Section 5.1.1.1) is also necessary.

The actual version of *Lúthien* has been tested on a 2-GHz Intel Pentium Core Duo notebook PC with 2 gigabytes of RAM. However, it is not necessary to use a high-performance system. For large-scaled FITS images processing the only real requirement is sufficient amount of the available operating memory – at least 1 gigabyte of RAM memory is recommended.

The software runs on Linux platform - primarily developed under (K)Ubuntu 7.04, a free, Debian derived Linux-based operating system. It should also run on other major *nix distributions flawlessly.

A.3 Distribution Information

The software distribution can be found on the enclosed CD-ROM. Following listing shows the content of the distribution:

- ▷ `thesis.pdf` – Electronic version of this document.
- ▷ `prog/src/` – Directory containing the actual implementation of the *Lúthien* software package.
- ▷ `prog/dependencies/` – Directory containing necessary software dependencies distributions - RTS2 library, CFITSIO, libnova and ImageMagick.

- ▷ `output_fits/` – Directory for the output of the software.
- ▷ `test_fits/` – Directory containing example input data.
- ▷ `documentation/` – Directory containing *Lúthien* documentation generated from code comments using Doxygen and CVS development statistics generated by StatCVS.
- ▷ `INSTALL` – Text file containing the installation instructions.
- ▷ `README` – Text file containing the notes on how to use the software.

A.4 Running Lúthien

The input of the *Lúthien* application is an image pair of two consequent FITS files of the same size, and various parameters. The output is a list of detected lines in the input images and, optionally, input images with detected lines positions marked. The following section presents the software options and shows some examples of usage.

The console command to run the program is as follows

```
$ luthien {-t TYPE|-e|-r} -i IN1:IN2 [-o OUT1[:OUT2]]
                                         [-m MASK] [-n NUM] [-v]
```

- ▷ `TYPE` – type of the HT pre-processing pipeline (possible values 1 and 2). First one analyzes egdels (detected by Sobel detector), the latter analyzes the image data itself.
- ▷ `IN1, IN2` – The input image names including paths. Both images have to be FITS files (including image data HDU) of the same size dimensions. It is crucial that the processed input images were taken in a small time window, as one of the basic ideas in *Lúthien* implementation is to compute their pixel-by-pixel *difference* image.
- ▷ `OUT1`, `OUT2` – The output image names including paths. The detected lines will be drawn on the input images background. In case only one output name is specified, all the detected lines are will be drawn on the first input image background.
- ▷ `MASK` – The mask image name including path. The mask cannot be of resolution smaller than the input images.
- ▷ `NUM` – The offset for input images data loading. Default value is '*1*'.

The detection method specification and the input filenames are obligatory.

If the output filenames are specified, the result will be saved as images of the given type. For image conversion the ImageMagick library is used (also IM, see Section C.2). Thus, all the output image formats supported by IM are available, including DPX, EXR, GIF, JPEG, JPEG-2000, PDF, PhotoCD, PNG, Postscript, SVG, and TIFF.

In case the mask image name is specified, the input images will be masked using the mask image data and operation of multiplication. Hence, mask has typically pixels of two brightness values – zeroes at positions to be masked out, and 1's at positions where the actual processed data lies. The mask cannot be of resolution smaller than the input images. Again, the mask file can be of any image format supported by IM. As the border areas of astrography images often contain details disturbing the line detection, it is highly recommended to mask them out.

Use the '`-n`' option, if you want to open the FITS files in a special opening mode, when only every `NUM`-th pixel is loaded. This may result in great computational and memory savings. On the other hand, the detection may not yield correct results. Recommended value is '`1`' (default value) or '`2`'.

The full list of options with brief desription:

Option	**Description**
`-help` | `--help`	Get a complete list of options.
`-i IN1:IN2`	Names of two input images separated by ':'.
`-o OUT1[:OUT2]`	Save output to the image file(s) with name(s) `OUT1` and `OUT2` (including extension). In case only one name is specified, all the detected lines are saved using the first input image as background. [*optional*]
`-m MASK`	Use masking with a mask with name `MASK`. [*optional*]
`-t TYPE`	Use the Hough transform method for line detection. TYPE specifies one of two pre-processing pipelines (possible values '1' and '2').
`-e`	Use the edge detection method for line detection.
`-r`	Use the region growing method for line detection. One of the detection methods has to be specified.
`-n NUM`	Set the offset for input images data loading to `NUM`. Default value is '`1`'. [*optional*]
`-v`	Verbose text output.

Examples of use

This command takes image files `A.fits` and `B.fits` from directory `img` and analyzes the image data using the edge detection method:

```
$ ./luthien -e -i img/A.fits:img/B.fits
```

The following command takes image files `A.fits` and `B.fits` from directory `img`, loads every second pixel of them, masks the data with `mask.png`, analyzes the data using the Hough transform method and saves the results in directory `img_out` as `A_HT.png` and `B_HT.png`:

```
$ ./luthien -t 2 -n 2 -i img/A.fits:img/B.fits
                          -o img_out/A_HT.png:img_out/B_HT.png -m mask.png
```

To see brief help and option usage, use:

```
$ ./luthien -help
```

A.4.1 Configuration File

The astrographic images processed by *Lúthien* can vary in many aspects and so do the desired results of the detection method. To achieve higher flexibility of used detection approaches, a number of important program constants is configurable in the `config.cfg` text file. Setting these constants according to the actual conditions may considerably improve quality of the output. Their summary including description is presented in this section.

The *overall* settings used by more than one detection method are as follows

fitsLoadIncStepX – Offset for loading image data from a FITS file in the x-direction. Default value: 2.

fitsLoadIncStepY – Offset for loading image data from a FITS file in the y-direction. Default value: 2.

lineMinPixelLength – Minimum length of detected line. Default value: 75.

linesNoMax – Maximum number of returned detected lines. Default value: 30.

drawLinesNoMax – Maximum number of detected lines drawn in the output images. Default value: 10.

blobElongatednessComputeStepsNo – Number of rotation steps for blob elongatedness computation. Default value: 20.

maxPixelValue – Maximum pixel brightness value in the image. Default value: 65535.

modeSaveTempImages – True if temporary results should be saved as images to disc. Default value: *false*.

The *Hough transform method*-related settings are

HTpreprocessThresholdRatio The threshold for image preprocessing stage in HT method is computed as `(imageMaxValue * HTpreprocessThresholdRatio)`. Default value: 0.4.

HTsobelThreshold – Threshold for filtering of Sobel edge detector results. Default value: 0.3.

HTdilateIterationsNo – Number of dilation iterations in the morphology stage of HT method. Default value: 1.

HTdilateMaskSize – `(HTdilateMaskSize x HTdilateMaskSize)` is the the size of a square dilation structuring element. Default value: 3.

HTerodeIterationsNo – Number of erosion iterations in the morphology stage of HT method. Default value: 1.

HTerodeMaskSize – `(HTerodeMaskSize x HTerodeMaskSize)` is the size of a square erosion structuring element. Default value: 3.

HTlineThickness – Approximate thickness of lines detected. Used for Hough space accumulator resolution computation. Default value: 10.

HTlineMaxGap – Maximum line gap for the line verify stage of HT method. Default value: 5.

HTpeaksThresholdSensitivityFactor – The threshold for peak detection is `(sqrt {number of black pixels} * HTPeaksThresholdSensitivityFactor / FitsLoadIncStepX)`. Default value: 15.

HTpeaksThresholdMin – Minimum number of votes for a peak to be accepted in the peak detection stage. Default value: 100.

HTpeaksThresholdMaxSteps – Maximum number of peak detection iterations. Default value: 10.

The *region growing method*-related settings are

RGpreprocessThresholdRatio The threshold for image preprocessing stage in RG method is computed as `(imageMaxValue * RGpreprocessThresholdRatio)`. Default value: 0.6.

RGforegroundThresholdAddTolerance – Minimum brightness value for a pixel to become part of a blob is `(blob seed pixel value - foregroundThresholdTolerance)` (first of the conditions). Default value: 20 000.

RGforegroundThresholdMin – Minimum brightness value for a pixel to become part of a blob is `(max pixel value * foregroundThresholdAbsMin)` (second of the conditions). Default value: 0.45.

RGblobMinLength – Minimum length (in pixels) of a blob to be accepted in the blob verify stage. Default value: 40.

RGblobMinElongatedness – Minimum elongatedness of a blob to be accepted in the blob verify stage. Default value: 4.5.

RGblobMinSize – Minimum size (in pixels) of a blob to be accepted in the blob verify stage. Default value: 200.

RGblobMaxSize – Maximum size (in pixels) of a blob to be accepted in the blob verify stage. Default value: 15000.

The *edge detection method*-related settings are

EDcannyLowThreshold – Low hysteresis threshold for the Canny edge detector. Default value: 0.2.

EDcannyHighThreshold – High hysteresis threshold for the Canny edge detector. Default value: 0.6.

EDblobMinSize – Minimum size (in pixels) of a blob to be accepted in the blob verify stage. Default value: 200.

A.4.2 Using Hough Transform Method

Probably the most accurate and robust of the three detection methods is the one based on the Hough tranform. This section presents the procedure in more detail and shows which constants (contained in the configuration file, see Section A.4.1) to tweak to get the desired detection results. The HT image processing pipeline is designed as follows:

1. Input images data is loaded – use option '`-n NUM`' to take only every `NUM`-th pixel or specify variables `fitsLoadIncStepX` and `fitsLoadIncStepY` to do the same in the x-and y-direction, respectively.

2. The data is equalized and composed using subtract operation into one data array.

3. At this moment, there are two possible pipelines to choose from. The first one computes the Sobel edge detection, thresholds and dilates its output (using a square structuring element of size `(HTdilateMaskSize x HTdilateMaskSize)`) to get a 'sobel mask image'. The original difference data is thresholded and masked with this sobel mask. The regular mask image is then (optionally) applied. The second pipeline masks out and thresholds the difference image. In the morphology stage, the data is converted to binary and inverted. Dilation with a square structuring element of size `(HTdilateMaskSize x HTdilateMaskSize)` is performed `HTdilateIterationsNo` times. Erosion with a square structuring element of size `(HTerodeMaskSize x HTerodeMaskSize)` is performed `HTerodeIterationsNo` times. The histogram of difference data is not ideally bipolar, but the brightness difference between the foreground (meteorites, stars) and background (dark sky) is usually sufficient for choosing a good threshold value – then the second method works well enough. But sometimes one of the input images is strongly affected (for instance, by reflectors of a car), causing too big difference in the input images. In such cases, the first method performs much better.

4. The result data is thinned.

5. The Standard Hough transform follows. The quantization of the parameter space (and hence, the overall accuracy) depends on the `HTlineThickness` constant. Maximum number of detected lines is given by `linesNoMax`.

6. The initial peak detection threshold (means minimum number of votes for a line to be accepted) is determined by `HTPeaksThresholdSensitivityFactor` and `HTPeaksThresholdMin`. The peak detection is repeated until a desired number of peaks is detected (less than `linesNoMax`), but maximum of `HTPeaksThresholdMaxSteps` times.

7. Consequently, the detected lines are verified. Only a line of Euclidean length `of lineMinPixelLength` pixels (or more) is accepted. For robustness, gaps shorter than `HTLineMaxGap` do not disrupt a line.

8. The line filtering stage then finds similar lines (in slope and intercept), throws out duplicates and connects parts of the same line.

9. In the final (optional) step, the found lines are drawn on the original input data and the result images are saved (the output filenames are set using option '`-o`').

A typical command to use the Hough tranform method would look like this

```
$ ./luthien -t 1 -n 2 -i img/A.fits:img/B.fits
                        -o img_out/A_HT.png:img_out/B_HT.png -m mask.png
```

The program takes image files `A.fits` and `B.fits` from directory `img`, loads every second pixel of them, masks the data with `mask.png`, analyzes the data using the Hough transform method and saves the results in directory `img_out` as `A_HT.png` and `B_HT.png`.

A.4.3 Using Edge Detection Method

The second detection pipeline is based on the Canny edge detector technique. This section presents the procedure in more detail and shows which constants (contained in the configuration file, see Section A.4.1) to tweak to get the desired detection results. The edge detection method (ED) looks as follows:

1. Input images data is loaded – use option '`-n NUM`' to take only every `NUM`-th pixel or specify variables `fitsLoadIncStepX` and `fitsLoadIncStepY` to do the same in the x-and y-direction, respectively.

2. The data is equalized, composed using subtract operation into one data array and masked out (optional step - see option '`-m`').

3. Edge detection using the Canny edge detector is performed. The low and high threshold values for the hysteresis step are determined by `EDcannyLowThreshold` and `EDcannyHighThreshold`.

4. In the morphology stage the data is converted to binary and inverted. Dilation with a square structuring element of size `(HTdilateMaskSize x HTdilateMaskSize)` is performed. Erosion with a square structuring element of size `(HTerodeMaskSize x HTerodeMaskSize)` is performed.

5. Blobs of bright pixels are detected using simple flood-fill algorithm. Only blobs satisfying given conditions are accepted – minimum pixel size at least

`EDblobMinSize` pixels, minimum Euclidean length of the blob at least `RGblobMinLength` and minimal elongatedness of `RGblobMinElongatedness` or more.

6. The line filtering stage then finds similar lines (in slope and intercept), throws out duplicates and connects parts of the same line.

7. In the final (optional) step, the found lines are drawn on the original input data and the result images are saved (the output filenames are set using option '`-o`').

A typical command to use the Hough tranform method would look like this

```
$ ./luthien -e -n 2 -i img/A.fits:img/B.fits
                          -o img_out/A_HT.png:img_out/B_HT.png -m mask.png
```

The program takes image files `A.fits` and `B.fits` from directory `img`, loads every second pixel of them, masks the data with `mask.png`, analyzes the data using the ED method and saves the results in directory `img_out` as `A_HT.png` and `B_HT.png`.

A.4.4 Using Region Growing Method

The final detection technique is based on simple region growing. This section presents the procedure in more detail and shows which constants (contained in the configuration file, see Section A.4.1) to tweak to get the desired detection results. The region growing method (RG) looks as follows:

1. Input images data is loaded – use option '`-n NUM`' to take only every `NUM`-th pixel or specify variables `fitsLoadIncStepX` and `fitsLoadIncStepY` to do the same in the x-and y-direction, respectively.

2. The data is equalized, composed using subtract operation into one data array, masked out (optional step - see option '`-m`') and thresholded (see constant `RGpreprocessThresholdRatio`).

3. In the morphology stage the data is converted to binary and inverted. Dilation with a square structuring element of size `(HTdilateMaskSize x HTdilateMaskSize)` is performed. Erosion with a square structuring element of size `(HTerodeMaskSize x HTerodeMaskSize)` is performed.

4. Blobs of bright star pixels are detected using simple region growing method. The thresholded data gives us the seed pixels position. A neighbouring pixel

is marked as a part of the blob, if its brightness value is higher than (max pixel value * RGforegroundThresholdMin) and not lower than (seed value - foregroundThresholdTolerance). Moreover, only blobs satisfying further conditions are accepted – minimum pixel size at least RGblobMinSize pixels, minimum Euclidean length of the blob at least RGblobMinLength and minimal elongatedness of RGblobMinElongatedness or more.

5. The line filtering stage then finds similar lines (in slope and intercept), throws out duplicates and connects parts of the same line.
6. In the final (optional) step, the found lines are drawn on the original input data and the result images are saved (the output filenames are set using option '-o').

A typical command to use the Hough tranform method would look like this

```
$ ./luthien -r -n 2 -i img/A.fits:img/B.fits
                           -o img_out/A_HT.png:img_out/B_HT.png -m mask.png
```

The program takes image files A.fits and B.fits from directory img, loads every second pixel of them, masks the data with mask.png, analyzes the data using the RG method and saves the results in directory img_out as A_HT.png and B_HT.png.

Appendix B

Lúthien - Developer's Manual

Lúthien is an integrated software package designated to process high-resolution wide-field FITS images in order to detect meteorite trails. This appendix is a developer's manual for version 1.0 of the package.

Author

Igor Gomboš

Feedback

Please direct any comments or suggestions about this document to:
`luthien.project@gmail.com`.

Acknowledgements

This documentation was created in LaTeX (`http://www.latex-project.org/`) using the LyX document processor (`http://www.lyx.org/`) .

Version

First edition. Published 14 December 2007.

Related documentation

User's Manual to the *Lúthien* software package supplied as part of the distribution.

B.1 What is Lúthien?

Lúthien is an integrated software package designated to process high-resolution wide-field FITS images in order to detect meteorite trails. It implements three different detection methods based on the Hough transform, simple region growing and Canny edge detector.

It was developed in C++ programming language and runs on Linux platform. The software uses fragments of the RTS2 library package, software package designed for full robotic automatization of astronomical observations, created by Mgr. Petr Kubánek (for further details, see Section 5.2).

The detector's output can be saved as images of various formats. For image output, the ImageMagick library routines are employed.

It is strongly recommended to get familiar with the Lúthien User's Manual, as well. Not all of the user-related information is part of this document, as it is highly redundant.

B.2 Documentation Guidelines

For code documentation purposes, we decided to use a documentation generation system *Doxygen*. An on-line documentation (in HTML) and an off-line reference manual (in LaTeX) was generated from the documented source files of the project. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages.

The documentation is extracted directly from the sources. Hence, it is easy to keep the documentation consistent with the source code. In Doxygen, there are several ways to mark a comment block – JavaDoc style, Qt style, using C++ comment lines, etc. We decided to use the JavaDoc style which consists of a C-style comment block starting with two *'s, like this

```
/**
 * Brief description ending at first at this dot.
 * More detailed description follows here.
 * @param a an integer argument.
 * @param s a constant character pointer.
 * @see Test()
 * @see publicVar()
 * @return The test results
 */
```

To document members of a file, struct, union, class, or enum, it is sometimes desired to place the documentation block after the member instead of before. It then looks like this

```
TVal1, /**< enum value TVal1. */
```

For more information about Doxygen, see [16].

B.3 Coding Guidelines

To keep the code well-arranged and readable, it is desirable to follow some coding standard. We decided to get inspired by a list of C++ coding recommendations common in the C++ development community located at [25].

The recommendations are based on established standards collected from a number of sources, individual experience, local needs, as well as suggestions.

Source code should always be considered larger than the IDE it is developed within and should be written in a way that maximizes its readability independently on any particular IDE.

The list deals with general recommendations, naming conventions, statements, layout and comments, etc. It is strongly recommended to be aware of them and to use them where possible.

B.4 Components

The source code of *Lúthien* can be divided into three logical groups:

- ▷ *Algorithms* – Implementation of the algorithmic part of the package – various detection methods (Hough transform, Canny edge detection, simple region growing), basic image enhancement routines, morphological and segmentation techniques. The source code is located in the `prog/src/algorithms/` directory.

- ▷ *Geometry* – Implementation of the supporting geometric 2-D primitives and their manipulation methods – point and parameter, line, blob, etc. The source code is located in the `prog/src/geometry/` directory.

- ▷ *Other* – General purpose code of configuration settings management, timer for time complexity measurement, C++ macros, central image data management, application main function, etc. The source code is located in the `prog/src/` directory.

These groups will now be discussed in further detail in the following section.

B.5 Code Reference

This section is rather an overview of the methods and their location, not an exhaustive description of every source code line. For further information, see the documentation generated by Doxygen and the commented code itself which are both part of the distribution.

Algorithms

- ▷ `methodHough.h + .cc`, `methodEdgeDetection.h + .cc`, `methodRegionGrowing.h + .cc` – Include functions implementing the three detection methods of *Lúthien*. In general, they can be divided into certain stages, namely loading data, preprocessing stage, morphology stage, core detection, line filtration and image saving.

- ▷ `hough_sht.h + .cc` – Implementation of the Standard Hough transform, as described in [17]. Normal line parametrization is used. Included function `verifyLines` filters the previously detected lines based on some conditions (minimal length, gap).

- ▷ `houghSpace.h + .cc` – Class `L_HoughSpace` for a 2-D Hough space (accumulator array) for HT line detection. Methods for memory allocation, cell incrementation (`incrementAccum`), quantization computation (`computeQuantization`) are supplied. `findLines` detects peaks in the accumulator array using local maxima or butterfly method, `computeLineParameters` finds line parameters (starting and ending point, votes number, etc.) .

- ▷ `imageEnhancement.h + .cc` – Collection of some basic image enhancement and manipulation methods - image composition, histogram equalization, convolution, thresholding, smoothing with a box filter or weighted averaging. The class is implemented as a singleton object accesible anywhere in the package.

▷ `morphology.h + .cc` – Collection of some basic morphological methods - erosion and dilation with a square structuring element and thinning. The class `L_Morphology` is implemented as a singleton object.

▷ `segmentation.h + .cc` – Class `L_Segmentation` (also a singleton) implements a collection of some basic segmentation methods, e.g., Canny detection, Sobel edge detector, blobs growing. `filterLines` filters line detection results, finds similar lines (in slope and intercept), throws out duplicates and connects parts of the same line.

Geometry

▷ `geometry.h + .cc` – Various geometry-related functions. `computeIntersectionPoints` finds intersection points of a line with an image, `getLineBresenhamCoords` generates line point coordinates between given start and end points using the Bresenham algorithm, etc.

▷ `point2D.h` – Template class for 2-D points (`L_Point2D`).

▷ `param2D.h` – `L_Param2D` is designed as a template class for 2-D parameter. It is derived from the `L_Point2D` class.

▷ `line2D.h` – Structure `L_Line` provides a simple line representation. Includes some special attributes for use with the Hough transform. `L_LineVotesComparator` is used for line comparison based on the number of votes in the accumulator cell of HT.

▷ `array2D.h` – Template class `L_Array2D` implements a 2-D array with some basic manipulation methods.

▷ `blob.h + .cc` – Class `L_Blob` implements a vector of neighbouring pixels (of `L_Point2D<int>` type) and their boundary, with some special methods for region growing. Method computeElongatedness finds the maximum elongatedness value of the blob and other parameters. Class `L_BlobList` represents a list of `L_Blob` objects.

Other

▷ `configFile.h + .cc` – A C++ class for reading configuration files, developed by Richard J. Wagner.

- ▷ `dataVault.h + .cc` – Class `DataVault` stores important image data (input files, mask, temporary data) in an easily accesible way, implemented as a singleton object. Method `loadData` reads the input data specified with command line options.

- ▷ `fitsHelper.h + .cc` – A few routines to convert one image format to another (`convertFile`), to get copy of a FITS file (`copyFitsFile`) and to print FITS manipulation error message (`printFitsError`).

- ▷ `l_rts2AppImage.h + .cc` – Contains the main application class `L_Rts2AppImage`, derived from `Rts2AppImage` (in RTS2). Provides command line options processing and starts the actual line detection using `revealThemLuthien` mehod.

- ▷ `l_rts2Image.h + .cc` – `L_Rts2Image` is a wrapper class for FITS file manipulation. It is derived from `Rts2Image` (RTS2 library).

- ▷ painter`.h + .cc` – Class drawing lines into output images and saving them via *ImageMagick* routines.

- ▷ `settings.h + .cc` – Implements a class providing access to the main application settings read from config file. `Settings` is implemented as a singleton object. Method `parseConfig` parses the configuration file '`config.cfg`' and saves the setting values.

- ▷ `singleton.h` – `Singleton` is a template class implementing the Singleton pattern (for further details, see [63]).

- ▷ `timer.h + .cc` – Group of functions for precise time measurement (down to 1/1000 of a second).

- ▷ `utils.h` – Simple macros for rounding and comparing numbers, conversion between degrees and radians, some constants definition.

- ▷ `config.cfg` – Configuration file containing application parameter specification.

Appendix C

Resources

C.1 Developing Resources

The C++ code was developed using *Netbeans* ([32]) and *Eclipse* ([31]) IDE versions for C/C++ development. The LaTeX code of this work was written in *LyX* document processor ([66]). For the bibliography management, *JabRef* ([37]) appeared to be a good choice. For version control, *CVS* support integrated in the Eclipse IDE was taken advantage of.

Some of the images were created in *Ipe*, the extensible drawing editor ([36]), and *Adobe Photoshop* ([64]).

Documentation was generated by *Doxygen* ([16]), a documentation system for C++, C, Java, Objective-C, Python, etc. Statistical HTML report from the CVS repository was generated by StatCVS.

C.2 Used Packages

For FITS files manipulation, the *CFITSIO* library is used (see Section 5.1.1.1). The *RTS2* software package (see Section 5.2) and *ImageMagick* library (see Section C.2) are taken use of, as well.

"ImageMagick is a software suite to create, edit, and compose bitmap images. It can read, convert and write images in a variety of formats (over 100) including DPX, EXR, GIF, JPEG, JPEG-2000, PDF, PhotoCD, PNG, Postscript, SVG, and TIFF. The functionality of ImageMagick is utilized through object-oriented C++ API called *Magick++*. ImageMagick is free software, whose license is compatible with the GPL. It runs on all major operating systems", due to [35].

C.3 Third-party Code

Some parts of the code (e.g., timer implementation, thinning algorithm in segmentation and the `L_Point2D` class) are based on the code of the *Image Processing Library 98* (IPL98) developed by René Dencker Eriksen, distributed under LGPL licence. For more information, see [1].

The Canny detector method implementation is based on the code of the *Open Source Computer Vision Library* [47] distributed under Intel License Agreement For Open Source Computer Vision Library [48].

Configuration File Reader for C++ (the `ConfigFile` class) is a simple open-source code distributed under MIT License, created by Richard J. Wagner [12].

Appendix D

Configuration File

```
##########################################################
#### Configuration file for Luthien software package ####
##########################################################
#
# Comments are denoted by '#' and run till the end of line.
#
# This file should be located at $(PREFIX)/config.cfg, where
# $(PREFIX) is the standard prefix passed to the ./configure script
#
# Values names and values are separated by '='
# Numerical values are handled as numbers,
# string values which are handled as strings.
#
# Reasonable default values are set here and in the code itself.
# Not passing a value in the config file means the default value will
# be used. Be careful, because the default values may not fit your needs.
#
# Examples of use:
#
# apples = 7             # comment after apples
# price  = 1.99          # comment after price
# sale   = true          # comment after sale
# title  = one fine day  # comment after title
# weight = 2.5 kg        # comment after weight
# zone   = 1 2 3  # comment after 1st point
#          4 5 6  # comment after 2nd point
#          7 8 9  # comment after 3rd point
#
```

```
# Igor Gombos, <luthien.project@gmail.com>

#=======================================
# ---- Overall settings
#=======================================
# Offset for loading image data from a FITS file in the x-direction.
# Only every fitsLoadIncStepX-th pixel in x-direction will be loaded.
# Default value: 1.
fitsLoadIncStepX = 1

# Offset for loading image data from a FITS file in the y-direction.
# Only every fitsLoadIncStepY-th pixel in y-direction will be loaded.
# Default value: 1.
fitsLoadIncStepY = 1

# Minimum length of detected line.
# Default value: 75
lineMinPixelLength = 75

# Maximum number of returned detected lines.
# Default value: 50
linesNoMax = 50

# Maximum number of detected lines drawn in the output images.
# Default value: 5
drawLinesNoMax = 5

# Number of rotation steps for blob elongatedness computation.
# Default value: 20
blobElongatednessComputeStepsNo = 20

# Maximum pixel brightness value in the image.
# Default value: 65535
maxPixelValue = 65535

# In common greyscale images 255
# True if temporary results should be saved as images to disc.
# Default value: false
modeSaveTempImages = false
```

```
# True if mask image should be used for results computation.
# Default value: true
flagUseImageMask = true

#========================================
# ---- Hough Transform Method settings
#========================================
# The threshold for image preprocessing stage in HT method
# is computed as (imageMaxValue * HTpreprocessThresholdRatio).
# Default value: 0.4
HTpreprocessThresholdRatio = 0.4

# Number of dilation iterations in the morphology stage of HT method.
# Default value: 1
HTdilateIterationsNo = 1

# The size of a square dilation structuring element is
# (HTdilateMaskSize x HTdilateMaskSize).
# Default value: 3
HTdilateMaskSize = 3

# Number of erosion iterations in the morphology stage of HT method.
# Default value: 1
HTerodeIterationsNo = 1

# The size of a square erosion structuring element is
# (HTdilateMaskSize x HTdilateMaskSize).
# Default value: 3
HTerodeMaskSize = 3

# Approximate thickness of lines detected.
# Used for Hough space accumulator resolution computation.
# Default value: 10
HTlineThickness = 10

# Maximum line gap for the line verify stage of HT method.
# Default value: 5
HTlineMaxGap = 5

# The threshold for peak detection is computed as
# (sqrt(number of black pixels)
# * HTPeaksThresholdSensitivityFactor / FitsLoadIncStepX).
# Default value: 15
HTpeaksThresholdSensitivityFactor = 15

# Minimum number of votes for a peak to be accepted
# in the peak detection stage.
# Default value: 100
HTpeaksThresholdMin = 100

# Maximum number of peak detection iterations.
# Default value: 10
HTpeaksThresholdMaxSteps = 10

#========================================
# ---- Region Growing Method settings
#========================================
# The threshold for image preprocessing stage in RG method
# is computed as (imageMaxValue * RGpreprocessThresholdRatio).
# Default value: 0.5
RGpreprocessThresholdRatio = 0.5

# Minimum brightness value for a pixel to become part of
# a blob is (blob seed pixel value - foregroundThresholdTolerance)
# (first of the conditions).
# Default value: 10000
RGforegroundThresholdAddTolerance = 10000

# Minimum brightness value for a pixel to become part of
# a blob is (max pixel value * foregroundThresholdAbsMin)
# (second of the conditions).
# Default value: 0.3
RGforegroundThresholdMin = 0.3

# Minimum length (in pixels) of a blob to be accepted
# in the blob verify stage.
# Default value: 40
RGblobMinLength = 40

# similar to lineMinPixelLength
# Minimum elongatedness of a blob to be accepted
# in the blob verify stage.
# Default value: 4.5
RGblobMinElongatedness = 4.5

# Minimum size (in pixels) of a blob to be accepted
# in the blob verify stage.
# Default value: 200
RGblobMinSize = 200

# We are not alone. We never have been. There are people outside.
# People with telescopes ... big telescopes.
# With cameras, taking thousands and thousands of images. All the time.
# I met one of them this day. How can I help you?, I ask.
# Look, we have these photos, he says.
# Of the stars, quiet and beautiful. And other great things, too, he says.
# Like meteorites. They are somewhere, they have to be.
# We need you to find them.
#
# You with us?

#========================================
# ---- Edge Detection Method settings
#========================================
# Threshold for filtering of Sobel edge detector results.
# Default value: 100
EDsobelThreshold = 100

# Low hysteresis threshold for the Canne edge detector.
# Default value: 0.05
EDcannyLowThreshold = 0.05

# High hysteresis threshold for the Canne edge detector.
# Default value: 0.5
EDcannyHighThreshold = 0.5

# Minimum size (in pixels) of a blob to be accepted
# in the blob verify stage.
# Default value: 200
EDblobMinSize = 200
```

Bibliography

[1] The Image Processing Library 98. [online, http://www.mip.sdu.dk/ipl98], November 2007.

[2] SAOImage DS9: Astronomical Data Visualization Application. [online], http://hea-www.harvard.edu/saord/ds9/, December 2007.

[3] M. Atiquzzaman. Multiresolution hough transform-an efficient method of detecting patterns in images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(11):1090–1095, 1992.

[4] D.H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, Vol.13, No.2:p.111–122, 1981.

[5] C. M. Brown. Inherent bias and noise in the hough transform. *IEEE Trans. Patt. Anal. Machine Intell*, vol. PAMI-5, no. 5:pp. 493–505, 1983.

[6] J. Brian Burns, Allen R. Hanson, and Edward M. Riseman. Extracting straight lines. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(4):425–455, 1986.

[7] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.

[8] CCFITS. [online], http://heasarc.gsfc.nasa.gov/docs/software/fitsio/CCfits/, December 2007.

[9] Vassilios Chatzis and Ioannis Pitas. Fuzzy cell hough transform for curve detection.

[10] Vassilios Chatzis and Ioannis Pitas. Randomized fuzzy cell hough transform.

[11] H. Y. H. Chuang and C. C. Li. A systolic array processor for straight line detection by modified hough transform. *Proc. IEEE Comput. Soc. Workshop Comput. Architecture Patt, Anal. Image Database Mgmt.*, pages pp. 300–304, 1985.

[12] ConfigFile. [online], http://www-personal.umich.edu/ wagnerr/ConfigFile.html, September 2007.

[13] G. Cook and E. Delp. A gaussian mixture model for Edge-Enhanced images with application to sequential edge detection and linking. pages 540–544.

[14] G. W. Cook and Edward J. Delp. Multiresolution sequential edge linking. In *ICIP*, pages 41–44, 1995.

[15] Stanley R. Deans. *The Radon transform and some of its applications.* A Wiley-Interscience Publication. New York etc.: John Wiley and Sons. XI, 289 p., 1983.

[16] Doxygen. [online], www.doxygen.org, December 2007.

[17] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, 1972.

[18] A. J. Castro-Tirado et al. Bootes-ir: a robotic nir astronomical observatory devoted to follow-up of transient phenomena. *SPIE*, 6267, 2006.

[19] A. Etemadi. Robust segmentation of edge data. *Proc 4th IEE Intern Conf Image Processing and Applications, Maastricht*, 1992.

[20] CFITSIO Quick Start Guide FITSIO. [online], http://heasarc.gsfc.nasa.gov/docs/software/fitsio/quick/quick.html, December 2007.

[21] W. Frei and Chung-Ching Chen. Fast boundary detection: A generalization and a new algorithm. *IEEE Transactions on Computers*, 26(10):988–998, 1977.

[22] M.J.E. Golay. Hexagonal parallel pattern transformations. *IEEE Transactions on Computers*, 18(8):733–740, 1969.

[23] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2002.

[24] Hideaki GOTO and Hirotomo ASO. On the efficient sampling interval of the parameter in hough transform. *The transactions of the Institute of Electronics, Information and Communication Engineers*, 81(4):697–705, 19980425.

[25] C++ Programming Style Guidelines. [online], http://geosoft.no/development/cppstyle.html, December 2007.

[26] N. Guil, J. Villalba, and E.L. Zapata. A fast hough transform for segment detection. 4(11):1541–1548, Nov. 1995.

[27] K. Hanahara, T. Maruyama, and T. Uchiyama. A real-time processor for the hough transform. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(1):121–125, 1988.

[28] P. V. C. Hough. Methods and means for recognizing complex patterns. In *U.S. Patent 3 069 654*. 1962.

[29] R. Hudec. Wide-field sky monitoring - optical and x-rays. *Mem. S.A.It.*, Vol.74, 973, 2003.

[30] Galen C. Hunt and Randal C. Nelson. *Lineal Feature Extraction by Parallel Stick Growing*. 1996.

[31] Eclipse IDE. [online], http://www.eclipse.org, December 2007.

[32] NetBeans IDE. [online], http://www.netbeans.org, December 2007.

[33] J. Illingworth and J. Kittler. The adaptive hough transform. *IEEE Trans. Pattern Anal. Mach. Intell.*, 9(5):690–698, 1987.

[34] J. Illingworth and J. Kittler. A survey of the hough transform. *Comput. Vision Graph. Image Process.*, 44(1):87–116, 1988.

[35] ImageMagick. [online], http://www.imagemagick.org, November 2007.

[36] Ipe. [online], http://tclab.kaist.ac.kr/ipe, December 2007.

[37] JabRef. [online], http://jabref.sourceforge.net/index.php, December 2007.

[38] H. Kalviainen. *Randomized Hough Transform : New Extensions*. PhD thesis, Lappeenranta University, 1994.

[39] S. Kamat, V. Ganesan. A robust hough transform technique for description of multiple linesegments in an image. *International Conference on Image Processing*, 1:216–220, 1998.

[40] Y. Kanazawa and K. Kanatani. Optimal line fitting and reliability evaluation. *IEICE Trans. Inf. 4 Syst.*, E79-D-9:1317–1322, 1996.

[41] N. Kiryati, Y. Eldar, and A. M. Bruckstein. A probabilistic hough transform. *Pattern Recogn.*, 24(4):303–316, 1991.

[42] Jelínek M. Vítek S. de Ugarte Postigo A. Nekola M. & French J. Kubánek, P. Rts2: a powerful robotic observatory manager. *SPIE*, 6274, 2006.

[43] E. Oja L. Xu and P. Kultanen. A new curve detection method: Randomized hough transform (rht). In *Pattern Recognition Letters*, 1990.

[44] J.-W. Lee and I.-S. Kweon. Extraction of line features in a noisy image. *Pattern Recognition*, vol. 30, no. 10:pp. 1651–1660, 1997.

[45] S. Lefevre, C. Dixon, C. Jeusse, and N. Vincent. A local approach for fast line detection. In *Proc. 14th International Conference on Digital Signal Processing DSP 2002*, volume 2, pages 1109–1112, 1–3 July 2002.

[46] Hungwen Li, Mark A Lavin, and Ronald J Le Master. Fast hough transform: A hierarchical approach. *Comput. Vision Graph. Image Process.*, 36(2-3):139–161, 1986.

[47] Open Source Computer Vision Library. [online], http://www.intel.com/technology/computing/opencv, November 2007.

[48] Open Source Computer Vision Library License. [online], http://www.intel.com/technology/computing/opencv/license.htm, December 2007.

[49] Sheng Liu, Charles F. Babbs, and Edward J. Delp. Line detection using a spatial characteristic model.

[50] David G. Lowe. *Perceptual Organization and Visual Recognition.* Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[51] A. Witkins M. Kass and Terzopoulos. Snakes: active contour models. In *Int. Journal Computer Vision*, 1988.

[52] R. Boyle M. Sonka, V. Hlavac. *Image Processing, Analysis, and Machine Vision, 2nd ed.* PWS Publishing, Pacific Grove, CA, 1999.

[53] Abdul-Reza Mansouri, Alfred S. Malowany, and Martin D. Levine. Line detection in digital pictures: a hypothesis prediction/verification pardigm. *Comput. Vision Graph. Image Process.*, 40(1):95–114, 1987.

[54] M. Mattavelli, V. Noel, and E. Amaldi. A new approach for fast line detection based on combinatorial optimization.

[55] M. Morimoto, S. Akamatsu, and Y. Suenaga. A high-resolution hough transform using variable filter. pages III:280–284, 1992.

[56] Kazuhito Murakami and Tadashi Naruse. High speed line detection by hough transform in local area. *icpr*, 03:3471, 2000.

[57] Vishvjit S. Nalwa and Eric Pauchon. Edgel-aggregation and edge-description. *Comput. Vision Graph. Image Process.*, 40(1):79–94, 1987.

[58] A General Package of Software to Manipulate FITS Files NASA's software FTOOLS. [online], http://heasarc.gsfc.nasa.gov/docs/software/ftools/ftools_menu.html, December 2007.

[59] R. C. Nelson. Finding line segments by stick growing. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(5):519–523, 1994.

[60] R. Nevatia and K. R. Babu. Linear feature extraction and description. In *Computer Vision Graphics and Image Processing*, 1980.

[61] Wayne Niblack and Dragutin Petkovic. On improving the accuracy of the hough transform. *Mach. Vision Appl.*, 3(2):87–106, 1990.

[62] K. Koral P. H. Eichel, E. J. Delp and A. J. Buda. A method for a fully automatic definition of coronary arterial edges from cineangiograms. *IEEE Trans. Med. Imag.*, vol. 7:pp. 313–320, 1988.

[63] Singleton Pattern. [online], http://en.wikipedia.org/wiki/Singleton_pattern, December 2007.

[64] Adobe Photoshop. [online], http://www.adobe.com/products/photoshop/index.html, December 2007.

[65] J. Princen, J. Illingworth, and J. Kittler. A hierarchical approach to line extraction based on the hough transform. *CVGIP*, 52:57–77, 1990.

[66] LyX Document Processor. [online], http://www.lyx.org, December 2007.

[67] 2nd Version Remote Telescope System. [online], http://lascaux.asu.cas.cz/rts2/, December 2007.

[68] S. D. Shapiro and A. Iannino. Geometric constructions for predicting hough transform performance. *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-1, no. 3:pp. 310–317, 1979.

[69] B. Shu, C. C. Li, J. F. Mancuso, and Y. N. Sun. A line extraction method for automated sem inspection of vlsi resist. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(1):117–120, 1988.

[70] NASA's software FITS Data Format. [online], http://heasarc.gsfc.nasa.gov/docs/heasarc/fits.html, December 2007.

[71] Jiqiang Song, Min Cai, Michael R. Lyu, and Shijie Cai. A new approach for line recognition in large-size images using hough transform. In *ICPR '02: Proceedings of the 16 th International Conference on Pattern Recognition (ICPR'02) Volume 1*, page 10033, Washington, DC, USA, 2002. IEEE Computer Society.

[72] R. L. Hartley A. Rosenfeld T. H. Hong, M. Shneier. Using pyramids to detect good continuation. *IEEE Trans. on Systems, Man and Cybernetics*, Vol. SMC-13, No.4:pp631–635, July / August 1983.

[73] A Brief Introduction to FITS. [online], http://fits.gsfc.nasa.gov/fits_intro.html, December 2007.

[74] T.M. van Veen and F.C.A. Groen. Discretization errors in the hough transform. 14(1-6):137–145, 1981.

[75] Wikipedia. Hough transform. [online], http://en.wikipedia.org/wiki/Hough_transform, November 2007.

[76] Lei Xu and Erkki Oja. Randomized hough transform (rht): basic mechanisms, algorithms, and computational complexities. *CVGIP: Image Underst.*, 57(2):131–154, 1993.

[77] Jiří Žára, Bedřich Beneš, Jiří Sochor, and Petr Felkel. *Moderní počítačová grafika.* Computer Press, Brno, 2004.

[78] Y. T. Zhou, V. Venkateswar, and R. Chellappa. Edge detection and linear feature extraction using a 2-d random field model. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(1):84–95, 1989.

[79] S.W. Zucker, R.A. Hummel, and A. Rosenfeld. An application of relaxation labeling to line and curve enhancement. *IEEE Transactions on Computers*, 26(4):394–403, 1977.