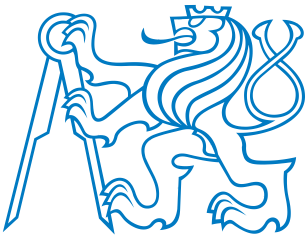
Czech Technical University in Prague
Faculty of Civil Engineering

Master's thesis

Prague 2012

Bc. Václav Petráš

Czech Technical University in Prague
Faculty of Civil Engineering
Branch Geoinformatics



Master's thesis

Building detection from aerial images in GRASS GIS environment

Detekce budov na leteckých snímcích v prostředí systému GRASS GIS

Supervisor: Ing. Ivana Hlaváčová, Ph.D.
Department of Mapping and Cartography

Prague 2012

Bc. Václav Petráš

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební
Thákurova 7, 166 29 Praha 6

ZADÁNÍ DIPLOMOVÉ PRÁCE

studijní program: Geodézie a kartografie
studijní obor: Geoinformatika
akademický rok: 2012/2013

Jméno a příjmení diplomanta: Bc. Václav Petráš
Zadávající katedra: Katedra mapování a kartografie
Vedoucí diplomové práce: Ing. Ivana Hlaváčová, Ph.D.
Název diplomové práce: Detekce budov na leteckých snímcích v prostředí systému GRASS GIS
Název diplomové práce v anglickém jazyce: Building detection from aerial images in GRASS GIS environment

Rámcový obsah diplomové práce: Cílem diplomové práce je vytvořit nástroje pro automatickou detekci budov a použít tyto nástroje pro detekci farem na leteckých snímcích severní Itálie. Nástroje budou vyvinuty v prostředí geografického informačního systému GRASS GIS. Díky tomu bude možné využít také nástroje, které tento free a open source GIS poskytuje. Detekce budov se zpravidla skládá z částí obecných, které jsou pro různé způsoby detekce, a z částí, které jsou specifické pro danou oblast a dané budovy. Ty části, které jsou obecné budou publikovány jako moduly systému GRASS GIS.

Datum zadání diplomové práce: _____ Termín odevzdání: 21. 12. 2012
(vyplňte poslední den výuky přísl. semestru)

Diplomovou práci lze zapsat, kromě oboru A, v letním i zimním semestru.

Pokud student neodevzdal diplomovou práci v určeném termínu, tuto skutečnost předem písemně zdůvodnil a omluva byla děkanem uznána, stanoví děkan studentovi náhradní termín odevzdání diplomové práce. Pokud se však student řádně neomluvil nebo omluva nebyla děkanem uznána, může si student zapsat diplomovou práci podruhé. Studentovi, který při opakovaném zápisu diplomovou práci neodevzdal v určeném termínu a tuto skutečnost řádně neomluvil nebo omluva nebyla děkanem uznána, se ukončuje studium podle § 56 zákona o VŠ č.111/1998 (SZŘ ČVUT čl 21, odst. 4).

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

original submission paper here

vedoucí diplomové práce vedoucí katedry

Zadání diplomové práce převzal dne: _____

.....
diplomant

Formulář nutno vyhotovit ve 3 výtiscích – 1x katedra, 1x diplomant, 1x studijní odd. (zašle katedra)

Nejpozději do konce 2. týdne výuky v semestru odešle katedra 1 kopii zadání DP na studijní oddělení a provede zápis údajů týkajících se DP do databáze KOS.
DP zadává katedra nejpozději 1. týden semestru, v němž má student DP zapsanou.
(Směrnice děkana pro realizaci stud. programů a SZZ na FSv ČVUT čl. 5, odst. 7)

Abstract

Despite the current availability of various geographic data, building detection is still needed under some circumstances. These circumstances can include inaccessibility of cadastral data, unrecorded and illegal building detection and disaster mapping. A number of approaches has been developed to detect buildings which rely on different kinds of spatial and quality features such as shape or color in remotely sensed imagery. The key properties differ among the various types of buildings (family house, factory, animal farm) and areas (city, rural area, state or country). This often requires the development of specialized detection algorithms. This work presents an algorithm developed to detect animal farm buildings in northern Italy. However, this implementation is intended to address not only the detection of farm buildings but to provide general tools for building detection in GRASS GIS. This work presents the implementation of an edge detector, line segment extractor and classification tool as well as the detection made on northern Italy orthophotos. All tools were implemented in the environment of the free and open source GIS software GRASS GIS. The use of free and open source software ensures that source code can be peer-reviewed and the analysis is highly reproducible.

Key words: GRASS, GIS, imagery, programming, C, C++, Python, building detection, free software, open source

Abstrakt

Navzdory dostupnosti různých druhů geografických dat je detekce budov za určitých okolností stále potřebná i v dnešní době. Tyto okolnosti zahrnují například nedostupnost katastrálních dat, potřebu detekovat nelegální a v katastru nevedené budovy a mapování škod po katastrofách. Bylo již vyvinuto mnoho způsobů detekce budov, které vychází z rozličných vlastností zachycovaných dálkovým průzkumem země. Klíčové vlastnosti se liší pro různé typy budov (rodinné domy, továrny, zvířecí farmy) a oblasti (město, venkov, země či stát). V důsledku toho je proto často nutné vyvíjet specializované detekční algoritmy. Tato práce popisuje algoritmy vyvinuté a použité pro detekci budov farem v severní Itálii. I přes nutnou specializaci některých nástrojů pro detekci budov farem se popisovaná implementace snaží snaží poskytnout i obecné nástroje pro detekci budov v GRASS GISu. Tato práce popisuje implementaci hranového detektoru, detektoru úseček, nástroje pro klasifikaci a dále pak detekci provedenou na ortofotech severní Itálie. Všechny nástroje byly implementovány v prostředí free a open source GIS softwaru GRASS GIS. Použití free a open source softwaru zajišťuje, že zdrojový kód může být přezkoumán a analýzy jsou dobře zreprodukovatelné.

Klíčová slova: GRASS, GIS, zpracování obrazu, programování, C, C++, Python, detekce budov, free software, open source

Declaration of authorship

I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged. Formulations and ideas taken from other sources are cited as such.

In Prague, December 20, 2012

(author's signature)

Acknowledgement

I would like to thank my family for the support, especially my sister who provided my with her photograph.

I would like to kindly acknowledge Anna Kratochvílová for a great collaboration on school project [1] which preceded this work as well as for a precious help with the development of the *i.farm.detect.rectangles* module. Also, I would like to thank her for the provided support in my GRASS GIS software leaning.

I would like to express my gratitude to Ivana Hlaváčová, my supervisor, for the guidance and good advice about detection algorithms.

I wish to thank Markus Neteler for pointing me to useful tools and libraries such as mlpy library.

I also wish to thank Luca Delucchi for creating special Addons repository which was used to publish part of the implemented software.

This work was partially supported by the project "IZS VE 09/09 RasterVet" funded by the Italian Health Ministry.

Contents

Prologue **21**

Conventions used in this document **23**

1 Introduction **25**
1.1 Free and open source software . . . 25
1.2 GRASS GIS environment . . . 26
1.3 The need for building detection . . . 26
1.4 Building detection in general . . . 27
1.5 Study area and Italian farm detection use case . . . 29
1.5.1 Input data . . . 29
1.5.2 Roof types . . . 31

2 Methods **35**
2.1 Rectangle detection . . . 35
2.1.1 Canny edge detector . . . 35
2.1.2 Hough transform for line detection . . . 36
2.1.3 Building rectangles using profiles . . . 40
2.1.4 Creating rectangle from line segments . . . 42
2.2 Supporting methods . . . 45
2.2.1 Detection of vegetation and other unwanted areas . . . 45
2.2.2 Rectangle filtering . . . 46
2.2.3 Duplicates reduction . . . 47
2.3 Measuring the success . . . 49

3 GRASS module development **51**
3.1 General notes . . . 51
3.2 Language choice . . . 52
3.3 Speed improvements and optimization . . . 53
3.3.1 Improving *r.houghtransform* logic . . . 53
3.3.2 Improving C++ code . . . 53
3.3.3 Module call overhead issue . . . 54
3.3.4 Improving Python code . . . 58

4 Results **61**
4.1 Implemented software . . . 61
4.1.1 General modules . . . 61
4.1.2 Modules specific for farm detection . . . 64
4.2 Application to Northern Italy images . . . 67
4.2.1 Detection procedure . . . 68
4.2.2 Results of detection . . . 68

Conclusion **73**

Bibliography **75**

A Installation guide **79**
A.1 Required components . . . 79
A.2 Installation . . . 79
A.2.1 GRASS 7 . . . 79
A.2.2 Modules from Addons . . . 79
A.2.3 Farm detection tools . . . 80
A.3 Testing of the installation . . . 80
A.3.1 Testing if pyGRASS works . . . 80
A.3.2 Testing if *i.edge* and *r.houghtransform* work . . . 81

B Input data import **83**
B.1 Approximate points import . . . 83
B.1.1 Shapefile with points . . . 83
B.1.2 CSV file with attributes . . . 83
B.1.3 Combining data coordinates and attributes . . . 84
B.2 Importing rasters in GeoTIFF format . . . 84
B.3 Training data import . . . 86
B.4 Formal input files description . . . 86
B.5 Unexpected attribute combination . . . 86

C User guide **91**
C.1 Map name conventions . . . 91
C.2 General process overview . . . 92
C.3 RGB to HIS conversion . . . 92
C.4 Smoothing and segmentation . . . 92
C.5 Texture . . . 92
C.6 Training areas . . . 92
C.7 Signature file . . . 92
C.8 Running the detection . . . 94
C.9 Associating rectangles and approximate points . . . 94
C.10 Testing the result . . . 96
C.11 Sample contents of the signature file . . . 96

List of Figures **99**

List of Tables **101**

List of Listings **103**

Prologue

Nowadays, there are many data sources available such as cadastral maps maintained by governments or OpenStreetMaps built up from various sources by a community. Despite this availability, we can encounter many situations when particular data is not available. This is often true for buildings. For example, it may not be possible to acquire cadastral data because of financial or legal reasons. On the other hand, there can be remote sensing data available such as aerial images and LiDAR data. In this case, building detection is the only way to obtain required information.

Many studies deal with building detection and several different approaches were proposed. Every detection method depends on available input data. Typical examples of input data include aerial images, multi-spectral aerial images and LiDAR data. The detection techniques range from supervised to fully automatic methods.

The motivation for this work is the Italian veterinary project — RasterVet — where farm buildings locations are needed. Since building locations are not available from cadastre or other maps, it is necessary to detect buildings from available data — color (RGB) orthophotos. Therefore, this work deals with farm building detection on orthophotos.

The workflow presented here is set up in GRASS GIS environment. GRASS GIS is a geographic information system providing wide functionality and modular environment. Furthermore, GRASS GIS is free and open source software. The same also applies to the newly implemented software. Hence, anyone can use this software as he or she likes; study and modify it and make use of his or her changes. Thanks to the use of free and open source software, the solution is reproducible which is an important characteristic of academic and research work.

This work is divided into four chapters. The first chapter introduces the motivation, related work, used GIS software and data available in the Italian farm detection use case. The second chapter describes the methods which were used and focuses on those which were newly implemented as GRASS GIS modules. The next chapter deals with GRASS GIS module development and describes specific implementation issues. The last chapter covers the achieved results namely implemented modules and its usage in Italian farm detection use case. Three appendices describe the ways to obtain and install implemented software, as well as its usage in Italian farm detection use case including the import of available data.

Conventions used in this document

This work mostly uses the term *image* or *aerial image* interchangeably with the term *orthophoto*[1] or *orthophoto mosaics*. Although there is a significant difference between these terms, the algorithms presented in this work are applicable to both. The term *raster map* refers to an georeferenced image in GIS software. A *binary raster map* or image is a image having only values zero and one.

Buildings and other objects in the image have edges. An *edge* is an rapid change in digital values. A pixel whose values are significantly lower or higher than pixels values around may represent a part edge.

There are several terms used in GRASS GIS which may be confusing probably because they were created before the currently used terminology has settled. The term *layer* usually refers to the GRASS layer which is a part of a vector map. The term *category* usually refers to the GRASS category which is close in meaning to imagery term class. However, GRASS uses categories also for vector maps. In this case the category is close in meaning to GIS term feature id. But in GRASS multiple features can have the same category. In this case the meaning is closest to the imagery term class or just to general term category. Note that GRASS has also term *feature id* but this id is usually used only internally.

If this document mentions *command line* (or command in command line) without any further specification, this command line is expected to be GNU/Linux command line or any similar Unix-like command line as available on Mac OS X, BSD and others. Scripts for command line are referred as *shell scripts*. Note for MS Windows users: Standard MS Windows command line does not provide functionality expected by this work. However, you can access this functionality through GRASS 7 command line, Git Bash[2] or many other freely available tools. The *GRASS command line* is command line which provides standard commands and also GRASS modules.

Source code in the text and source code listings are shown in the type writer font (e.g., `int a = 42`). Directories are also shown in the type writer font (e.g., `home`). Directory separator in paths is a slash (`/`). As usually, single dot (`.`) denotes current directory and two dots denote (`..`) parent directory. Three dots (`...`) denote ellipsis and thus something like your path to files.

While reading this work you may note higher number of graphs, or diagrams if

[1] `http://en.wikipedia.org/wiki/Orthophoto`
[2] `http://msysgit.github.com/`

you prefer. I, as the author, believe that graph is an ideal expression method for many things as noted also by [2]. So, it would be unfortunate for the reader as well as for the author to use complicated descriptions when the graph can be used.

Figures, tables and listings are numbered within a chapter using a chapter number in the prefix. For appendices letters are used instead of numbers. Footnotes are numbered within a chapter.

Chapter 1

Introduction

This chapter provides reader with basic information related to the motivation of this work, the GRASS GIS environment and the Italian farm detection use case.

1.1 Free and open source software

The *free software* definition contains four essential freedoms [3] which originally come from Stallman [4]:

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and change it so it does your computing as you wish.
- The freedom to redistribute copies so you can help your neighbor.
- The freedom to distribute copies of your modified versions to others.

Note that access to source code is necessary to fulfill these freedoms. Consequently, *open source software* is a highly related term and this term is widely used as well as the term free software. However, it was argued that it does not mean the same as the free software [5]. Thus, this work prefers the term *free and open source software* (FOSS).

The advantage to the user is that free and open source software companies and projects do not use per-copy license models where the user needs to pay for a copy of an application but they use models where the user pays for new features, additional services etc. [6] Moreover, the user — if he or she wants — can influence the software development in the way he likes. Besides the obvious advantage to users this is also advantage for free and open source software companies and projects which can benefit from the ideas coming from a wide community [7].

Moreover, the free and open source software is the best option for academics and researchers because open source code ensures that the implemented software — an output of the research — can be revised, criticized and reused [8]. Furthermore, not only the newly created code should be open but also all used calculations from the environment where the new code was developed should be available including their

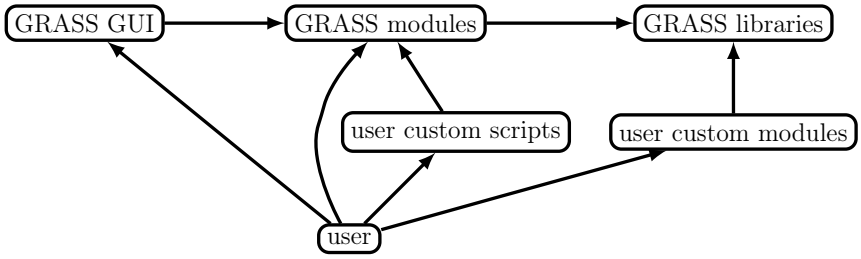


Figure 1.1: The basic ways a user can interact with the GRASS GIS functionality

source codes. This can be ensured by the incorporation of the new code into a free and open source project [9].

1.2 GRASS GIS environment

GRASS GIS [10] is a free and open source geographic information system (GIS). It is developed by various organization, companies and individuals from the early 80's [11]. GRASS GIS is often referred as GRASS.

The main characteristic of GRASS is that it is highly modular [10]. Modules are independent programs[1] and thus, GRASS is scriptable by nature. This means that user has the possibility to access GRASS functionality programmatically besides the usual way of accessing software functionality — graphical user interface (GUI), see figure 1.1 for details.

Users which are not familiar with the concept of command line programs and scripts can think about modules as individual tools. Furthermore, these users can still take an advantage of GRASS modules because they can access these tools through GUI and build scripts using graphical modeler without even knowing about command line. However, thanks to GRASS architecture, it is possible to smoothly and slowly switch to command line and scripts once advanced user realizes that he or she needs more power than any GUI can give. For criticism on modules as fully independent programs from developer point of view, see section 3.3.3.

1.3 The need for building detection

There are a lot of sources of spatial data available. Many of them contain buildings besides other features such as parcels, roads and land use. For example, in the Czech Republic there is a lot of data including buildings published [12] thanks to European directive INSPIRE[2] and other efforts. OpenStreetMap (OSM) [13] is a world wide

[1] GRASS module can be a written in C programming language as well as in C++, Python or Bash.

[2] `http://inspire.ec.europa.eu/`

freely licensed[3] data source which provides various types of data. The full list of data sources, of course, depends on the particular location.

Although many data sources for buildings exist, there are many occasions when it is necessary to create building polygons from measured data such as aerial images or LiDAR point clouds. These occasions may include disasters, illegal building detection, mapping in developing countries, the lack of building data in the dataset and a simple case of not having enough money to acquire vector building data.

In the Italian farm detection use case, we do not have access to cadastral data because of their price. Farm buildings are also not contained in maps accessible on the Internet such as OpenStreetMap or Google Maps. However, we have an access to RGB orthophotos and that's why RGB orthophotos are the input data in the Italian farm detection use case (section 1.5).

1.4 Building detection in general

Many studies deal with building detection and several different approaches were proposed. The main reason probably is that every detection method depends on available input data, on data type as well as on data quality and other features. Typical examples of input data include aerial images, multi-spectral aerial images and LiDAR data. Various approaches are listed here to provide an overview of related work. At the end, the method presented in this work is outlined. The success rates are not listed here since, as claimed by Shorter and Kasparis [14], it is hard to compare different methods because different authors use different datasets to test their methods. This corresponds to the fact that different data require different algorithms.

One possible approach is template or pattern matching. For building detection, the matching is required to be both rotation [15] and scale [16] invariant. Template matching based detections give good results when detecting clearly defined building shapes [17]. Karantzalos and Paragios [18] showed promising results and mentioned possibility of extending concept to 3D space. Rainsford and Mackaness [19] showed how different templates can be applied to find farm (rural) buildings. Unfortunately, this method would require a image preprocessing to be used on aerial image. The image had to be simplified (probably into a binary image) while the shape of the building had to be preserved which is hardly possible e.g., with building blending to parking lots (see the section 1.5.1).

The object based detection approaches address problems connected with vegetation [20], problems building roof complexity [21] and overall real scene complexity [22]. These methods usually make use of various segmentation[4] algorithms and constructs both 2D and 3D buildings. 2D building is usually only a building roof while 3D building has also wall and thus, a height. Other possibility is to construct building as a 3D object from its edges [23]. These methods can make use of overlapping

[3] OpenStreetMap's license — Open Data Commons Open Database License — allows copying, distribution and adaptation of OSM data as well as combining with other data, for details see `http://www.openstreetmap.org/copyright`.

[4] `http://en.wikipedia.org/wiki/Image_segmentation`

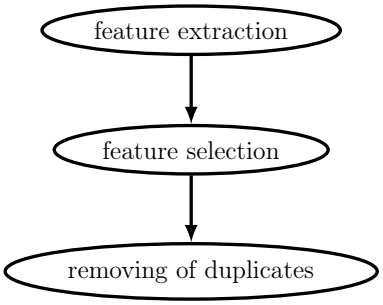


Figure 1.2: The simplified building detection workflow

aerial images and photogrammetric approaches [24].

The LiDAR (or Airborne Laser Scanning) is now often used for building detection. Although the heights obtained from LiDAR (elevation data) allow to detect buildings without additional information [25], the detection is often combined with aerial images when basic features or building candidates are evaluated or masked using these images [26, 27, 28]. Rutzinger et al. worked on object based building detection using LiDAR data in GRASS GIS environment [29, 30].

There are two things which are common to the majority of building detections. The first is the need for vegetation detection (which will be covered later). The second is the edge detection. Edges often represent borders of a building or another object and thus, give a reliable information about building position. However, subsequent steps are necessary to filter out edges or objects constructed from edges which do not represent buildings. The evaluation can be, for example, based on profiles, digital values in different channels, correlation between detected edges and a constructed object polygon, intersection or closeness with other features (e.g. known roads). Various methods can be, of course, combined. The methods how to decide if the object is a building varies form a simple scoring of various objects [31] to deciding based on graph theory [32].

The detection of vegetation, usually done by a classification of raster images, helps to avoid false detection and thus it is an important part of a building detection process. For instance, the big improvement in detection process is when an NIR (near-infrared) channel is available and so, it is possible to compute, for example, NDVI (normalized difference vegetation index). Some building detection methods can be even based only on classification of raster images. However, this would be not advantageous e.g., for farm building detection (section 1.5) because the number of building pixels on one image is low while the number of roads and parking lots is high. This would lead to high number of false positives (high branch factor, low correctness) as these algorithms show the tendency to classify roads and parking lots pixels as building pixels [14] because of the similarity in digital values between these areas and buildings.

This work presents a method which can be categorized as feature (object) detection (basic schema is in the figure 1.2). It is based on edge detection and subsequent line segment extraction which appears to be very general. Detected (complex) features — rectangles — are ranked and filtered in order to select (preserve) only those

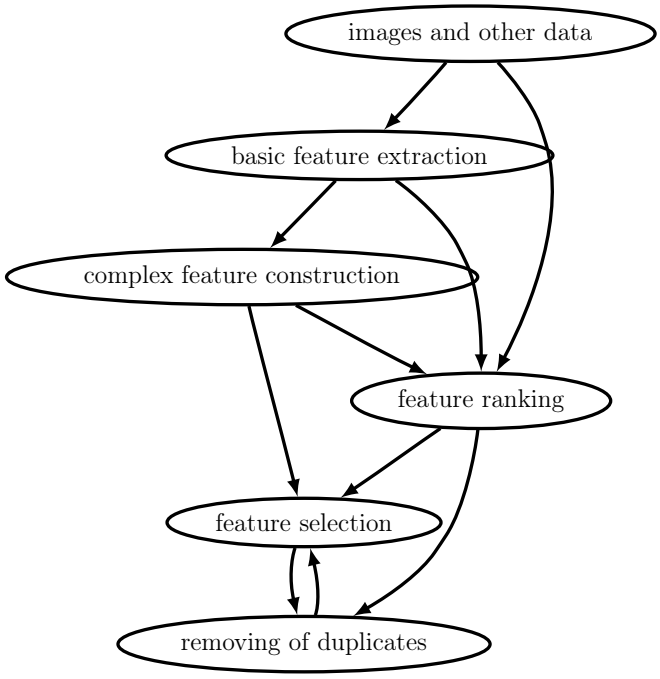


Figure 1.3: The general building detection workflow

representing buildings. The ranking is based on colors and thus, has to be specialized to particular area. The presented method is designed to automatically detect rectangular farm buildings on northern Italy orthophotos (see the section 1.5). A user needs to provide statistics[5] (his own or ones from this work) based on training areas to support detection of vegetation. Although, the method has to be highly specialized it was implemented in the way that individual steps can be replaced by different ones if they fit to the general template (figure 1.3) or reused in other methods if desired.

1.5 Study area and Italian farm detection use case

The described data are available for the Italian veterinary project — RasterVet — where farm buildings locations are needed. This section describes these input data and also basic analyses made on this data in order to suggest methods to use for building detection.

1.5.1 Input data

Input data contains georeferenced RGB (red, green and blue) orthophotos and approximate positions of farms. The difference for building detection in having orthophotos and having raw aerial images is not only the need for processing in case of aerial images but also the possibility to examine overlapping images to obtain

[5]The concept of GRASS signature file is used to convey statistics from one algorithm to another.

3D features from an image. However, in our farm detection use case, we have only orthophotos available, so the situation is similar to having only single nadir image.

The approximate points are a crucial part of input data. They allow us to search for the farm building only in the specific area. The size of this area was not specified but, according to the data examination, the area was defined as a rectangle with 100-meters-long sides. However, sometimes these points are badly positioned. They could be found in the barycenter of the farm, at the street entrance or at a place between buildings (see figure 1.4). This inconsistency is probably caused by the fact that these points were digitized by different persons and in different times. Therefore the size of a rectangle is subject to change.

(a) (b)

Figure 1.4: Approximate points are sometimes well positioned but sometimes they are far away from farm buildings which they belong to

The specified resolution of images (orthophotos) is of 0.5 m. Unfortunately, it must be noted that the experienced resolution of images is slightly worse, probably meters. The noise in images was removed but still some work can be done to improve images.

Several analyses were done on input images in order to obtain their basic features. Both RGB to HIS (hue, intensity, saturation) transformation and PCA (principal component analysis) were made (by *r.rgb.his* and *i.pca* modules). Computing the difference between the first PCA channel and the intensity channel showed that both channel are very similar. So, only HIS were used because in this case, it is possible at least to make assumptions about values since the meaning of the channels has clear interpretation.

Other analyses include texture extraction, value-based thresholding and classification. Many farms were surrounded by vineyards or other striped areas. These areas were well identified by correlation-based texture algorithm provided by the module *r.texture*.

The most successful way of getting information from images was SMAP (Sequential maximum a posteriori estimation) classification (provided by *i.gensigset* and *i.smap*) using not only HIS channels but also texture (see above). When enough

training areas were provided, this method was able to distinguish several basic classes such as trees and fields (vineyards). Although this classification was not capable to distinguish buildings directly, it was used as a part of the building detection to filter out the vegetation (see section 2.2.1).

Other classification and thresholding techniques were not so successful and were not used in developed algorithm. These include maximum-likelihood discriminant analysis (*i.maxlik*), segmentation (especially *r.seg*) and calculating a ratio between red and green channels using various coefficients in order to distinguish vegetation and green roofs (see below). However, it must be noted that the priority in the development was given to the implementation of edge-based building detection algorithms, so the time acquired for general detection was limited. Moreover, latest advances in the development of *i.segment* and *i.segment.xl* modules promise much better segmentation capabilities.

The farm is considered as a set of buildings with accessories such as parking lots. The examination of dataset showed that a farm can be represented by one building or by a set of buildings which may or may not be parallel to each other. The only additional object occurring together with a farm in significant amount of cases is a parking lot. However, parking lots have usually the same color as buildings (light gray) and are connected to these buildings. Hence, it is difficult to detect them separately e.g., using segmentation.

1.5.2 Roof types

In general, there are several ways to differentiate roof types. Roof type can be based on the shape, color, top view or objects placed on the roof. The third and fourth option is highly related to the purpose of the building. The term roof is in this context interchangeable with the term building because the roof is almost the only thing which is visible from the building.

Various roof features

All farm buildings are rectangular and they are characterized by the elongated shape. Hence, the presented detection procedure is focused only to this elongated shape. In theory, some other buildings which would have sufficiently long edges can be detected, too, however, this was not tested.

Some buildings have vents placed on the roof or chimney placed close to the wall. Although, this could help to distinguish various kinds of animals settled in a building, this was not examined further because of expected difficulty since these features are usually hard to see since they consist only of few pixels. Moreover, the information about animals is already associated with approximate points, so there is currently no practical need to determine the animal type form an image.

Visual and profile (in intensity channel) analysis showed that there are two basic types of building roof shapes. The first type is saddle-shaped roof and the second type is rounded-shape roof. Usage of this feature for building detection is described in the section 2.1.3. The profiles were examined using graphical interface for *r.profile* module.

The building shadow was not mentioned yet because it does not depend on the roof but the building itself. It is worth mentioning it since, generally speaking, it is a part of many detection algorithms (see the section 1.4). However, because the shadow is usually not large (farm buildings are low) and only some buildings have recognizable shadow (because of the building height or the illumination angle), no information about shadow was incorporated into the farm building detection.

Roof colors

The last and very important feature of buildings in the dataset is a roof color (figures 1.5 and 1.6). We can distinguish the following roof colors: light gray, dark (gray or brownish), red and green (usually connected with rounded-shape roof).

(a) (b)

Figure 1.5: The light roof type (a) and the dark one (b)

(a) (b)

Figure 1.6: The red roof type (a) and the green roof type together with light roofs (b)

The most common roof color in the sample dataset is a light gray (tables 1.1 and 1.2). Approximately 75% of the farm buildings have this color. The red roof is the

one with the smallest relative occurrence rate. Dark and green roofs occur almost equally.

Table 1.1: Absolute numbers and percentages of various colors of building roofs in the sample dataset

color	count	percentage
light	85	73.28 %
dark	12	10.34 %
red	6	5.17 %
green	13	11.21 %

Table 1.2: Percentages of cells and buildings areas in the sample dataset

color	roof cells	all cells
light	79.18 %	0.16 %
dark	7.97 %	0.02 %
red	4.05 %	0.01 %
green	8.79 %	0.02 %

By computing statistical values for each roof color as well as for other areas, it can be determined which values are typical for each roof color. These values can be used to distinguish a rectangle representing a building and a rectangle created from some random lines in fields (see the section 2.2.2).

Unfortunately, sometimes this is not enough. For example, for the red roofs it can be inferred that when using median of hue values one can distinguish red roofs from other roof types easily. However, it is necessary to distinguish them from other areas, and this is problematic because of extremely high standard deviation and interquartile range which is typical for non-building areas such as brown (and reddish) fields. It is also possible to leave this decisions on a computer using classification (again, see the section 2.2.2 for details).

Nevertheless, each roof color has some characteristic properties. These properties can be used to detect these buildings but since these properties differ, it is ideal to detect each roof type given by a color separately.

Since the allocated time was limited and the most common roof color was light gray (approximately 75%), I decided to focus detection on the light gray building roofs. However, only some steps in the final detection method depend on the color — detection was designed to be as general as possible — so many parts of the overall process can be used for all roof colors, especially the edge (and line segment) detection and detection based on profiles are color-independent. I also made an attempt to detect red roofs but the results are not yet sufficient and were not incorporated into this work.

Chapter 2

Methods

This part covers algorithms used in this work. Many of these algorithms are generally applicable to any detection problem. However, this part focuses on methods and algorithms used in the farm detection use case.

2.1 Rectangle detection

Rectangle detection has several phases, namely edge detection, line segment extraction and building construction. This work presents Canny edge detector, Hough transform as a line extraction algorithm and two possible ways of 2D building construction.

2.1.1 Canny edge detector

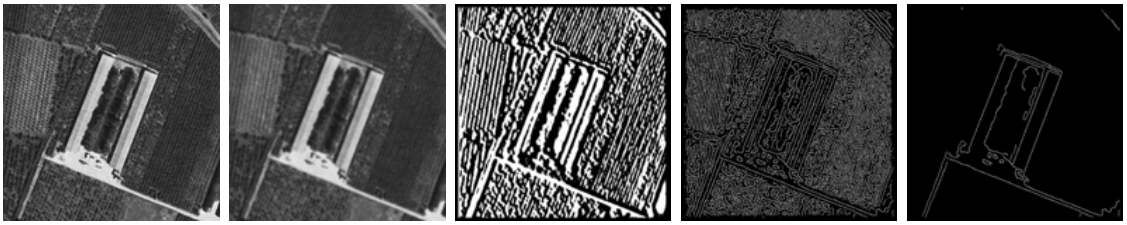
Edge is considered as a change in gradient which is computed from image digital values [33]. The Canny edge detector [34] is a filter which detects a wide range of edges in images. Although no usage statistics exist, we can say that Canny edge detector is well known and often used [35, 36, 37] not only to process images but also rasterized LiDAR data [38]. There are two main noticeable differences between Canny filter and other edge detectors. First, the others output broad lines (edges) while Canny filter outputs one-pixel-wide line which represents the most probable edge position [39]. Second, the Canny filter combines several steps together while other filters have only one step and often require some pre- or post-processing to get results which allows further processing. However, it must be noted that by applying subsequent filters, thresholding and edge thinning one can get similar results also from other edge detectors [33].

What is the desired output of edge detection depends on particular use. In case of building detection, it is often advantageous to get an image where edges are represented as thin lines (e.g. binary image) and that these edges are only the main edges in the image. The Canny edge detector gives us exactly this kind of output, so Canny filter is an ideal choice for many building detection algorithms.

Furthermore, Canny edge detector is considered as optimal edge detector according to these three criteria [40]:

1. important edges cannot be omitted and only actual edges can be detected as edges (no false positives);
2. difference in position of the real edge and the detected edge is minimal;
3. there is only one detected edge for an edge in original image (similar to first point).

The algorithm consists of a few steps [39] which are visualized on figure 2.1. Firstly, the noise is reduced by a Gaussian filter (based on normal distribution); the result is smoothed image. Secondly, two orthogonal gradient images are computed. These images are combined, so the final gradient can be defined by an angle and a magnitude (value). Next step is non-maximum suppression which preserves only pixels with magnitude higher than magnitude of other pixels in the direction (and the opposite direction) of gradient. Finally, only relevant or significant edges extracted by thresholding with hysteresis. This thresholding uses two constants; if a pixel magnitude is above the higher one, it is kept. Pixels with the magnitude under the lower constant are removed. Pixels with magnitude values between both constants are kept only when the pixels has some neighbor pixels with magnitude higher than the first constant [40].



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

Figure 2.1: Canny Edge Detection: (a) original image, (b) noise reduction, (c) intensity gradient image, (d) non-maximum suppression, (e) thresholding with hysteresis

The input is a gray scale image. Usually this gray scale image is an intensity channel obtained by RGB to HIS conversion. Some other possibilities include color edges (obtained from color channels) which may give slightly better results [41].

The output is typically a binary raster where ones denote edges and zeros denote everything else. There are also possible byproducts or intermediate products which can be part of the output, e.g. edge angles (gradient orientations) and edge strengths (gradient values). This method was implemented in the module *i.edge*.

2.1.2 Hough transform for line detection

In general, the Hough transform is a method for finding geometry structures in images. This work uses the Hough transform for straight line detection. Extracted

line segments can be used to construct rectangles or generally any other polygons. The transformation itself creates a raster image (figure 2.5 on page 39) which is then used for finding line segments. The input of Hough transform is an image which contains only edges. It is not necessary but for practical reasons, these edges should be thin edges e.g., those produced by Canny edge detector (2.1.1).

Lines can be mathematically represented in many ways. For this work, the following representation was chosen. Line is represented in polar coordinates (equation 2.1) where r is the distance between the line and the origin and θ is the angle of the vector from the origin to the closest point (see figure 2.2).

$$r = x \cos\theta + y \sin\theta \tag{2.1}$$

The point in the original image leads to a sinusoidal curve in the transformed

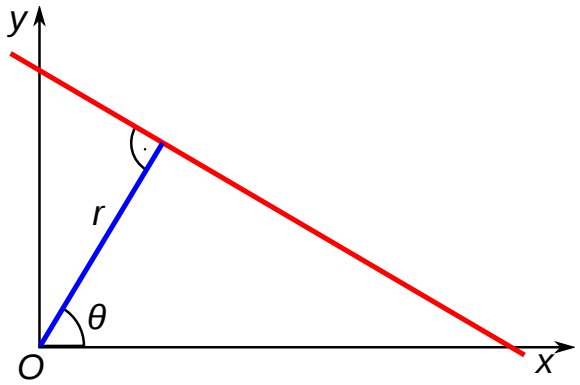


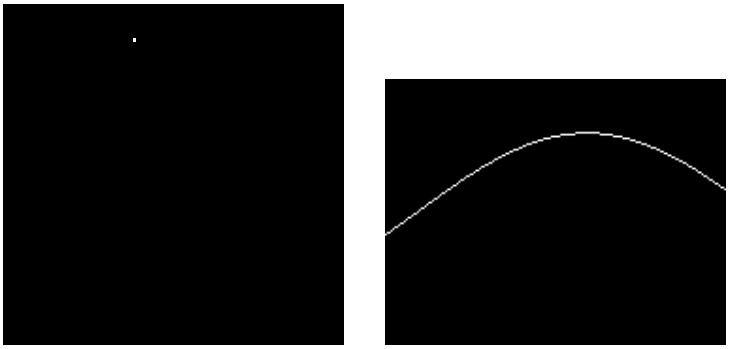
Figure 2.2: r, θ line parametrization

image. Points belonging to one line result in sinusoids intersecting in one point. The coordinates of this point describe the parameters r, θ of the line and its value represents the number of points of the line.

In other words, points from the original image with x and y axes are transformed into the Hough space with r and θ axes. We can say that the resulting image is the Hough image. One point (pixel) in the original image is represented by a curve in the Hough image (figure 2.3 on the following page a-b) and one point in the Hough image defines a line in the original image by r and θ parameters. One line in the original image is represented by an intersection of curves (figure 2.3 on the next page c-d). More points (pixels) in one line lead to higher value of the curves' intersection point.

For further evaluation, it is necessary to extract local maximum values from the Hough image which correspond to significant lines of the original image. This way we obtain the desired lines in form of (r, $theta$) coordinates. However, we do not obtain the end coordinates of the original line segment.

The trivial example of the original (artificial rectangle) and the transformed image is in the figure 2.4 on the following page. From the transformed image, we can infer certain rules and symmetries denoting a rectangle. There is an algorithm based on a moving window approach [42] which works well on this type of image. However, this algorithm is very slow due to the recurrent transformation [1]. Moreover, if we



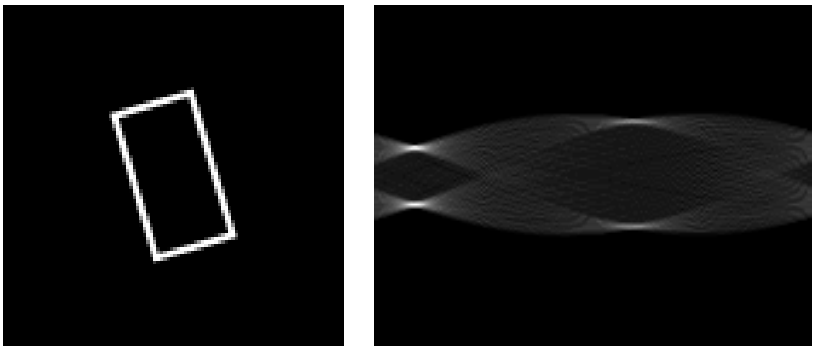
(a) Input image — one point

(b) Hough image — one curve

(c) Input image — points on one line

(d) Hough image — curves intersecting in one point

Figure 2.3: Explanation of Hough transform



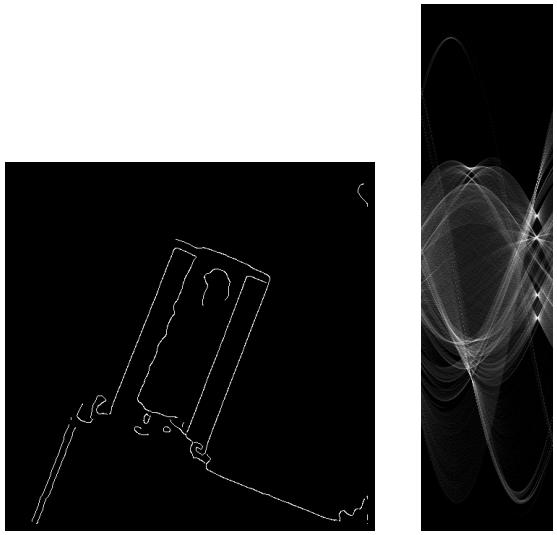
(a) raster rectangle (scaled)

(b) transformed raster

Figure 2.4: Basic Hough transform example showing recognizable features of a rectangle

look at the real image (figure 2.5 on the next page) of farm buildings (obtained by Canny edge detector), rectangles are not so clear as before. This is mainly caused by the fact that the shorter building side is less distinct or even not detected at all. Thus, different algorithm is needed to extract vector features from Hough image.

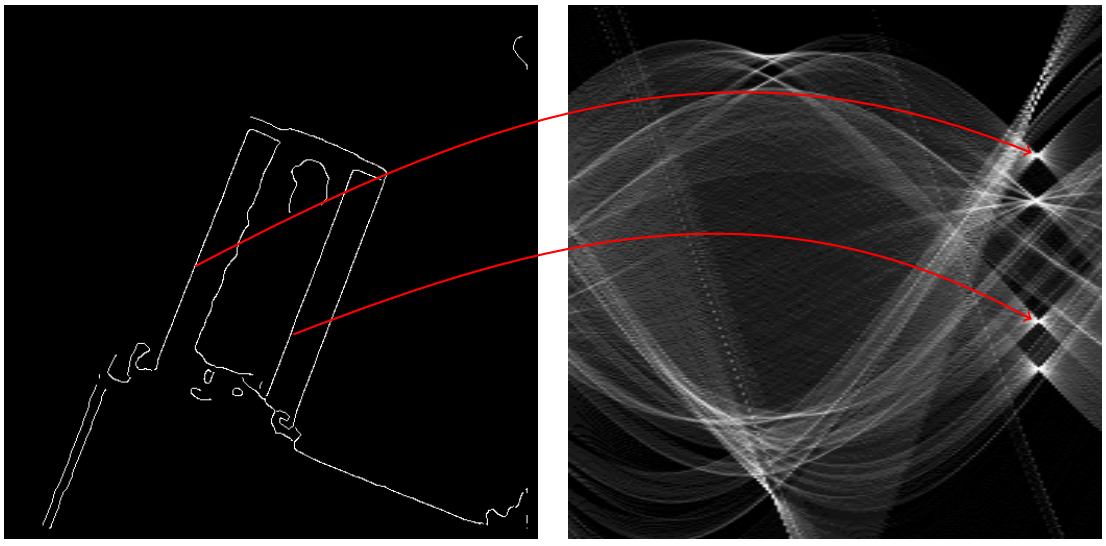
From the result of the Hough transform (figure 2.5 on the facing page) it is needed to extract the local maxima representing the lines. This is not a trivial task because



(a) (b)

Figure 2.5: Original image (a) and its transformation (b) with preserved relative scale

of existing noise in original image. The "identify and remove" Hough transform method described in [43] provides the actual coordinates of points (not just the line parameters) as a byproduct. It is based on the idea of sequential removing peaks from the transform result and eliminating the effect caused by the points of original image belonging to the removed peak (figure 2.6).



(a) (b)

Figure 2.6: Example of original and Hough transform image. Lines in the original image are connected with the corresponding maxima in Hough image

Lines obtained by "identify and remove" Hough transform had to be processed in order to get the actual line segments from the original image. Due to the noise in the original image, certain points can be included in the detected line although they do not belong to it. The subsequent step — extraction of line segments — has to ignore the outlying pixels. However, the main goal is to find separate line segments (pixels belonging to one line segment) so that the line segment is not interrupted. The method needs to tolerate also small gaps because some line segments can be interrupted, e.g. by vegetation. On the other hand, these gaps cannot be too frequent because many interruptions indicate line segments which are probably not part of the building. The serious line interruption means that there are two segments on one line in the image. As a result, the produced output can be more than one line segment for one detected line.

The Hough transform can be optimized by using additional information coming from the Canny edge detector computations or other edge detection algorithm. This additional information is directions of edges [44]. These angles are used to search only pixels which are in the direction of the particular line. The newly developed module *r.houghtransform* uses exactly the described approach.

It must be noted that some other works do not consider line segment reconstruction as a part of Hough transform. In fact, the basic result of Hough transform is the transformed image. However, it is necessary and also very advantageous for implementation of "identify and remove" Hough transform create data structures instead of an image. Furthermore, only the line segments are result which is ready to use. Considering these practical issues, Hough transform itself, its "identify and remove" extension and post-processing in order to get line segments are described together in this work.

2.1.3 Building rectangles using profiles

Generally, it is possible to create a profile by cutting vertically the surface created by image digital values as the image would be a digital elevation model and profile would be a terrain profile. Buildings have usually some characteristic profile, typically higher values, e.g. because of a higher intensity (in case of image) or a higher height (in case of elevation model). These characteristics can be visible in a profile and thus it can be stated if the profile corresponds with a building or not.

The obvious question is where to create a profile. It would be surely time consuming to create profiles in every direction on every place in the image. This work proposes to create two kinds of profiles whose position is based on the line segment obtained by Hough transform (section 2.1.2) or similar method. One kind is a perpendicular profile and the other is a parallel profile to the detected line segment. In the ideal case, examination of these profiles provides us with an information which edge is a part of the building and thus where the building is placed.

In order to get a complete information, the profiles are uniformly distributed along the line and its normal with relatively small distance, so that they cover the entire area of a possible farm building.

From perpendicular profiles a new profile is computed by using median values

(mean would not be robust enough). This profile (see figure 2.7) is then used for finding the position of the building according to its roof's shape. The shape depends on the type of the roof and the width of the building. Several kinds of templates representing various roofs ensures that all kind of roofs can be detected. The template is then matched with the perpendicular median profile using moving 1-dimensional window approach and computing correlation coefficient. Then the best matches are evaluated if they have meaningful position within the profile. From the position of the best match we can compute one pair of borders of the farm (parallel to the main direction of the farm).

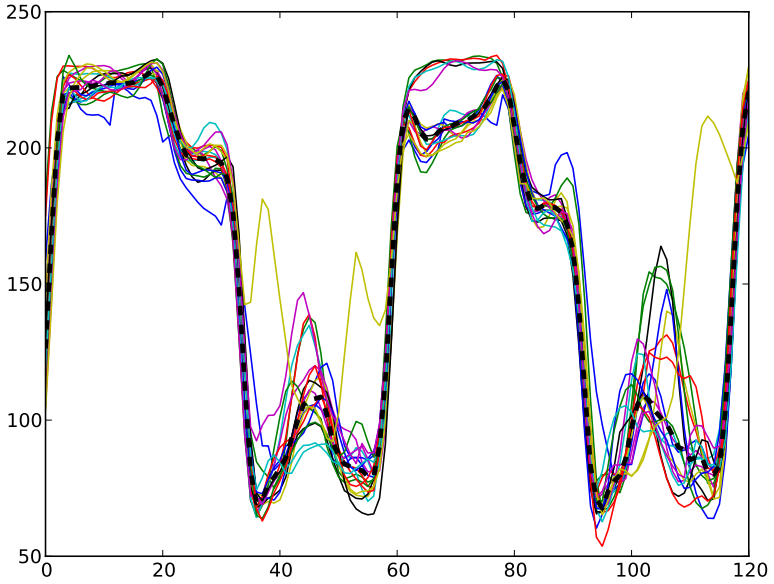


Figure 2.7: Perpendicular profiles with the highlighted median profile

Furthermore, it is required to find the other pair of the borders (representing the shorter sides of the farm). This is done by using the parallel profiles. Starting from an approximate center of the building, standard deviation is computed incrementally by adding values of the profile (see figure 2.8). It is assumed that the end of the building would mean a significant increase in standard deviation which can be then detected. This method based on standard deviation requires stability in the digital value and thus it is usable for long parallel profiles but not for perpendicular which require more time consuming correlation approach.

Having two perpendicular pairs of borders it is now possible to compute their intersection which produces four points representing the corners of the building.

The disadvantage is creation of a large number of profiles. This would be not necessary if perpendicular profiles would be used to detect short rectangle edges and parallel profiles would be used to detect long rectangle edges. Unfortunately, this would require less noisy data then are available in farm detection use case.

In conclusion, the input, in the case of farm detection, is an smoothed intensity image in order to get less noisy profiles. The output is a set of rectangles which

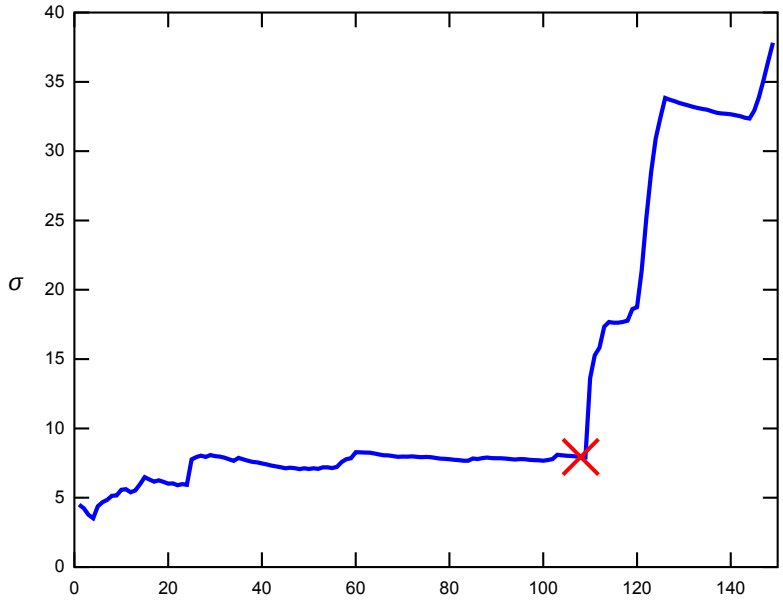


Figure 2.8: Standard deviation computed gradually from parallel profile values and indicated place where the significant value change occurs

can be, in the best case, used as they are without further processing. The output is characterized by a low number of false positives and low number of duplicates. However, the profile generation and processing is a time consuming task. Moreover, profiles assume that building is characterized by higher intensity values.

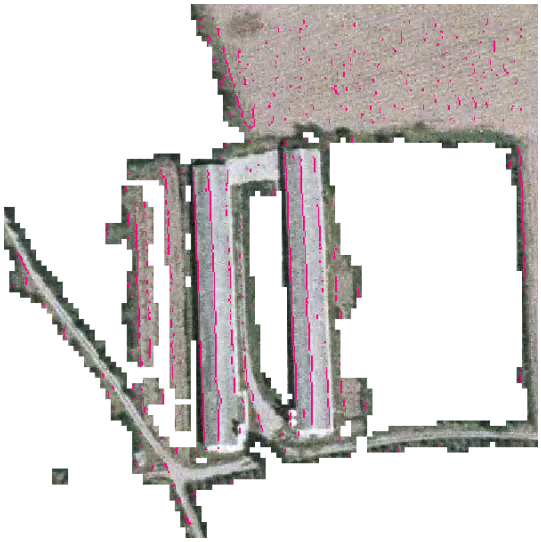
2.1.4 Creating rectangle from line segments

Line segments produced by Hough transform may or may not be a part of an building. To answer this question it is often necessary to have the rectangle first. Thus, this method only tries to generate all possible rectangles from a given set of line segments. So the result of this step is a set of rectangles [45] which needs to be subsequently evaluated to find those rectangles which represent buildings.

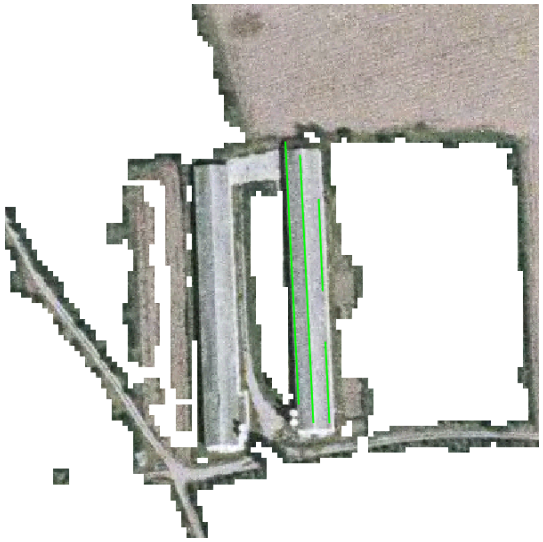
A rectangle is composed from two or more line segments which are obtained, in the case of farm detection, from several runs of Canny edge detector (*i.edge*) and Hough transform (*r.houghtransform*) (figure 2.9 on the next page).

More particularly, in the first step important line segments are extracted from image. In the second step, these (long) important line segments are used to find more line segments which would be hard to detect in a large image. To achieve different levels of details module *i.edge* is called with two different thresholds and σ value. The angle map (optional *i.edge* output) is reclassified in order to get parallel and perpendicular line segments. Using the line orientation, line segment length and distances between segments, only meaningful rectangles are created from parallel and perpendicular line segments. In farm detection use case, it is common that the

line segment perpendicular to the long line segment is not found. In this case missing short sides of the rectangle are computed from the two parallel line segments. The overview of basic steps is at the figure 2.10 on the following page.



(a) one group of detailed edges



(b) one group of longer parallel line segments

(c) one group of shorter perpendicular line segments

(d) all rectangles combined form line segments

Figure 2.9: Parallel and perpendicular line segments created from several edge maps are connected together in various combinations.

To reduce false positives in the early beginning, it is worth to use also a mask (see section 2.2.1) as an input. This mask can be used to filter out the edges in areas where no buildings are expected. This step can be done even after rectangle creation, however reducing number of edges and lines positively influences the speed of this method.

Described functionality is available through the *i.farm.detect.rectangles* module.

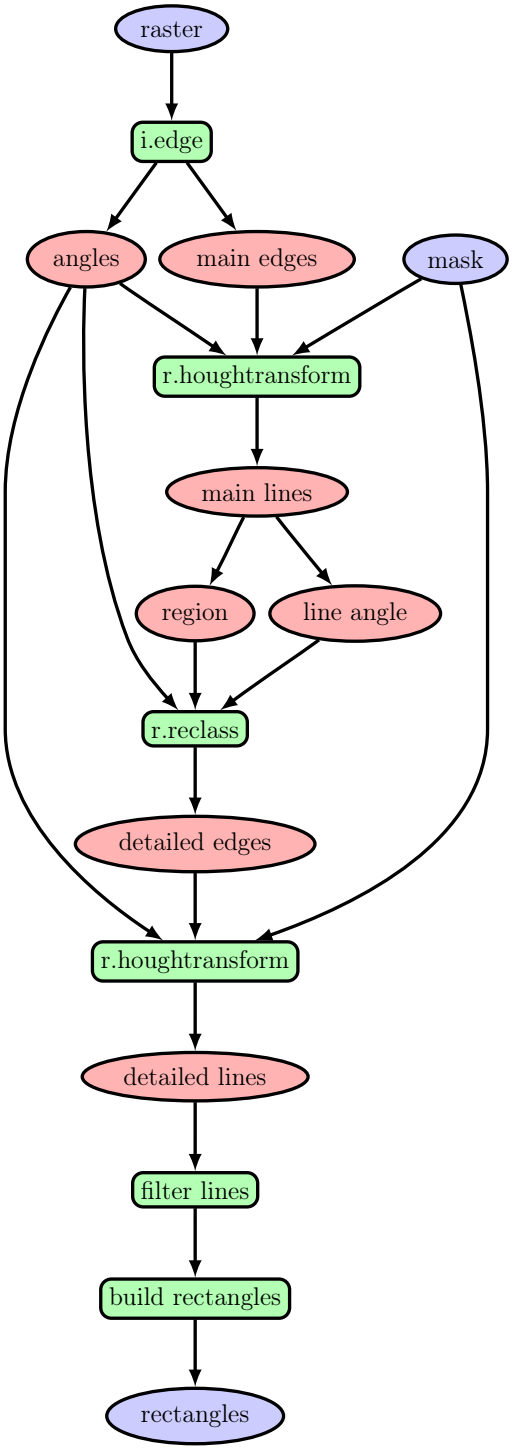


Figure 2.10: Simplified schema of rectangle detection (blue — input and output, red — other data, green — modules and functions; nodes are clickable)

2.2 Supporting methods

Building detection consists of many different steps besides the detection of edges and building polygons. This section covers steps which are performed after the detection of basic structures. However, some parts, especially the processing of raster maps, apply also to the preparation of the detection.

2.2.1 Detection of vegetation and other unwanted areas

Some of the areas in the image such as forests, fields and lakes can be excluded from building (rectangle) detection. Such areas can be detected using classification [33]. To distinguish this classification from classification of detected rectangles (mentioned below in the section 2.2.2) we can call this classification image or raster classification. Nevertheless, the underlying algorithm is the same and the required steps are similar. For unsupervised classification, which was used for farm building detection, manually or semi-automatically created training areas are necessary. Areas or pixels which are classified as for example, forests or fields, are the output of classification process.

Maps containing classified pixels or areas which are the output of classification can be used for computing statics for detected rectangles (section 2.2.2). The other possibility is to create "mask maps" — binary maps[1] where one value denotes the unwanted areas and the other denotes the areas designated for detection. In detection of rectangles, mask can limit the detection to meaningful areas (figure 2.11).

(a) map with applied mask (b) masked and unmasked edges

Figure 2.11: Mask used to filter out unwanted areas where there are no buildings (areas not covered by mask are excluded — standard GRASS GIS mask behaviour)

[1]GRASS uses mask which is a raster map where non-NULL and non-zero values denote the pixels where the computation will be performed. The mask can be called positive mask because values do not hide the pixels but non-values hide them.

It is worth mentioning that the input for the classification can consist of various raster maps, for instance hue, intensity and saturation channels (derived from RGB channels), textures (computed from intensity) or any other channel if it is available (infrared or near infrared channels usually significantly improve vegetation detection). Moreover, not only classification can be used to detect unwanted areas. Techniques like texture detection are able to output maps which can be used directly for mask creation. In any case, it is necessary to distinguish algorithms which create areas which can be surely excluded from building detection and algorithms which give only highly uncertain result. If we omit this difference, we can consequently omit many buildings during detection.

2.2.2 Rectangle filtering

The majority of rectangle detection algorithms produce both rectangles representing buildings and rectangles which are something different than buildings (false positives, see the section 2.3 for details). These rectangles do not represent buildings but roads, stripes in fields and other random, unwanted or even non-existing objects. The process of distinguishing good and bad matches can be called feature selection, evaluation, filtering or classification.

At first, it is necessary to specify criteria which will be used for evaluation. Typically, values coming from raster maps in the area of a rectangle are used and evaluated. Raster values are not used directly but some statistics such as mean and standard deviation are computed at first. Beside these statistics, rectangles can have also other characteristics coming from the rectangle detection (e.g., the correlation with profile when using the method from the section 2.1.3). In this step various data are collected, associated with a rectangle and prepared for the following classification or filtering.

One option is to use filtering based on empirical constants which are used to determine which rectangles represent buildings and which not. Although this method requires specification of particular values and so highly depends on the input data, it is easy to implement and fast when running because usually only a conditional expression needs to be evaluated on feature attribute data.

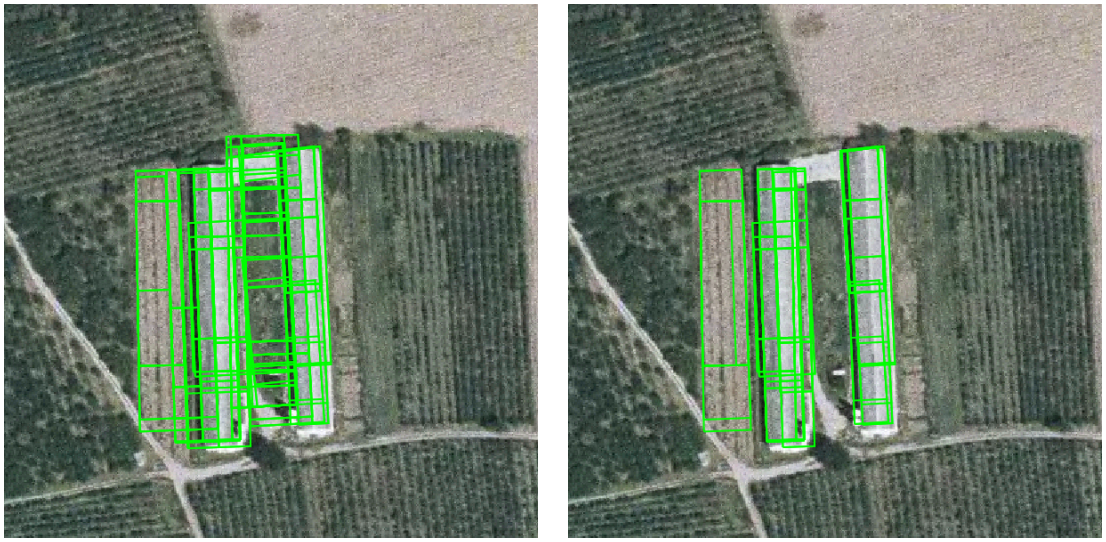
In our farm building use case, statistical values associated to each rectangle are obtained from raster maps. Empirical constants are based on the particular roof type (section 1.5). The obtaining the statistical values and the filtering itself are two separate steps.[2] The example of set of rectangles before and after filtering step is at the figure 2.12 on the facing page.

Other possibility is to use classification[3] based on machine learning[4]. The image classification serves to distinguish which image pixels or areas belong to which class. Here, the classification serves to distinguish rectangles which represent buildings

[2]Generally, it is possible to do both steps at once. However, considering the advantages of GRASS modular system, it is better to take advantage of existing modules and also create general modules. So, statistical values are obtained by the *i.farm.rast.stats* module and actual filtering is done by the *v.extract* module.

[3]`http://en.wikipedia.org/wiki/Statistical_classification`

[4]`http://en.wikipedia.org/wiki/http://en.wikipedia.org/wiki/Machine_learning`



(a) all detected rectangles (b) after light roof preserving filter

Figure 2.12: Filtering rectangles which are not buildings

and which represent something else. Classification is based on values associated with one rectangle[5] which are, for example, image (raster) digital values similarly to image classification. In farm building detection use case, a supervised classification was used. This classification requires manual or semi-automatic creation of training dataset which is the input to the learning phase of classification [33]. For computing itself, *mlpy* library [46] which provides various types of classifiers was used.

In farm building detection use case, manually digitized rectangles were used as training areas and another manually digitized dataset was used for the testing of classification. However, all digitized rectangles belong to one class (which can be called: *rectangle is a farm building*) and the training dataset has to contain data from all classes (*rectangle is a farm building*, *rectangle is not a farm building* and optionally also *rectangle may be a farm building or its part*). So a new training dataset was generated using the detected rectangles and the correct rectangles. The detected rectangles contain rectangles from all classes. The particular class is assigned to a detected rectangle using the information about positions of the correct rectangles. This approach is not limited only to our farm use case but it is applicable to any other building detection. The example of rectangles before and after classification is in the figure 2.13 on the next page.

2.2.3 Duplicates reduction

The removing of duplicate rectangle matches[6] is a necessary step for many buildings detection algorithms. The reason why duplicate matches are created in detection step is that building detection methods usually try to find all possible building

[5]Values or attributes associated with one feature used in classification are usually called properties or explanatory variables.

[6]In this case, a duplicate match means two or more overlapping rectangles.

(a) (b)

Figure 2.13: Rectangles before (a) and after (b) classification

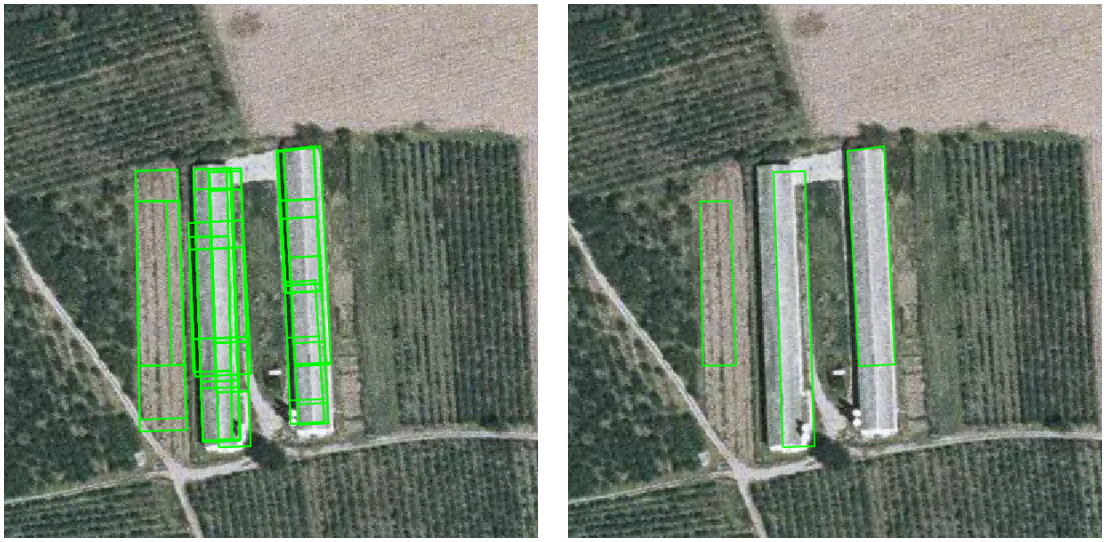
positions even if they overlap each other and so it is clear that only some of them are really buildings. The duplicates reduction step should select only the features (rectangles) which have higher probability to be the actual buildings. Whether this step is supposed to remove all duplicates or only some, depends on the phase when it is performed. If it is done before building (rectangle) filtering (section 2.2.2), it is sufficient when the algorithm removes only some duplicates because some duplicates can be removed during filtering. On the other hand, if this step is preserved after filtering, it should remove all duplicates unless the detection is semi-automatic and a user cleans the duplicates in the next step.

(a) overlapping rectangles which can be removed only by filtering

(b) duplicated rectangles caused by multiple detection of the southern farm

Figure 2.14: Different kinds of duplicates

As suggested above we can distinguish basically two kinds of duplicates. The first kind is rectangles which both represent something else than building. The second kind is rectangles when both represent a building (figure 2.14). The other kinds of duplicates are combination of two basic kinds duplicates. The decision making process in the case of farm building detection is based on statistical values associated with each rectangle (example result is at the figure 2.15 on the next page). Other



(a) rectangles after light roof preserving filter

(b) rectangles after removing duplicate matches

Figure 2.15: The example result of removing duplicates which were mainly based on intensity values

possibilities include duplicates removing based on size of rectangles and merging the rectangles together.

2.3 Measuring the success

There are several ways to rate detection. This work uses rating computed as detection percentage (DP) and branch factor (BF) according to Lin and Nevatia [45] (with some change in definitions explained below). Both values are percentages or decimal numbers. High DP values and low BF values denote higher success. DP is computed (2.2) from successfully detected object (true positives). BF is computed (2.3) from wrongly detected objects (false positives).

$$\mathrm{DP} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = \frac{\text{successfully detected objects}}{\text{actual number of objects}} \tag{2.2}$$

$$\mathrm{BF} = \frac{\mathrm{FP}}{\mathrm{TP} + \mathrm{FP}} = \frac{\text{wrongly detected objects}}{\text{number of detected objects}} \tag{2.3}$$

Where TP is the number of true positives, FP of false positives and FN the number of false negatives (i.e. not detected existing objects). These equations give the result in the interval [0, 1]. So, the numbers have to be multiplied by 100 to get actual percentage.

Definitions of true positives (TP), false positives (FP) and false negatives (FN) by Lin and Nevatia [45] and Müller and Zaum [47] are slightly different from generally used definitions. However, the meaning of equations is the same. Here is the overview of the terms TP, FP and FN as used in this work:

true positives (TP)	number of correctly detected objects
false positives (FP)	number of incorrectly detected objects
false negatives (FN)	number of omissions

There is no number for true negatives (TN) in the building detection when buildings are extracted as objects. We can say that this number would be something like a number of correctly detected non-objects. This number has obviously no meaning and therefore cannot be determined. In the classic evaluation of raster classification (with two or more classes e.g., buildings and non-buildings) this value or values are present. This is the main difference in measuring success rate by DP and BF values and by other methods.

The detection percentage is the same as completeness defined by Heipke et al. [48] for length of roads and applied to number of buildings by Shoter and Kasparis [14]. The same applies to branch factor which can be computed from correctness which is defined as ratio of correctly detected buildings and all detected buildings. Shoter and Kasparis [14] applied the same also to evaluation of classification of pixels. This in fact evaluates the accuracy in the position and shape of the building. However, since no requirements for position and shape precision were specified by the farm building detection project, the precision of position and shape (figure 2.16) was not exactly tested.

(a) (b)

Figure 2.16: Examples for rectangles being inaccurate in length. Correct rectangles are green, detected rectangles are red. We can see small length differences. Some rectangles are longer (a) while some other shorter (b).

Chapter 3

GRASS module development

This chapter contains general notes about GRASS GIS module development as well as the description of optimization of modules created within this work. The goal of this chapter is to give the reader a basic overview of the ways to access GRASS functionality in custom program and some of their advantages and disadvantages. Furthermore, particular methods, tools and design decisions are presented. At the end, the speed improvements and considerations are discussed. Everything is presented in the relation to modules developed within this work. In addition to that, an issue encountered when using module calls to do simple tasks is reported.

3.1 General notes

Modules for GRASS GIS can be written in C, C++ and Python programming languages. Another standard[1] way of writing new modules is to use Bash or another shell. This is for modules which mainly call only other modules.

Some functionality is accessible from GRASS libraries and some other through GRASS modules. The overview of various interfaces is in the figure 3.1 on the following page. The system is very flexible, so modules can use any interface (limitations are given only by the language). Note that presented overview (figure 3.1) is based on kinds of interfaces not on the functionality.

The module development itself is well documented by fully working examples[2] and by API documentation[3], moreover there are some additional guides on the Internet[4]not maintained by project members but its users.

Considering existing documentation, I will not write another tutorial, instead, I will describe tools which were used for the development. All these tools are free and open source, so this list can be a useful inspiration for any GRASS developer including students and professionals.

For C and C++ development (source code writing, debugging and profiling), I

[1] standard, i.e. supported in GRASS itself

[2] `http://trac.osgeo.org/grass/wiki/HowToProgram`

[3] `http://grass.osgeo.org/programming7/`

[4] `http://wiki.icess.ucsb.edu/rhessys/Developing_custom_GRASS_programs`

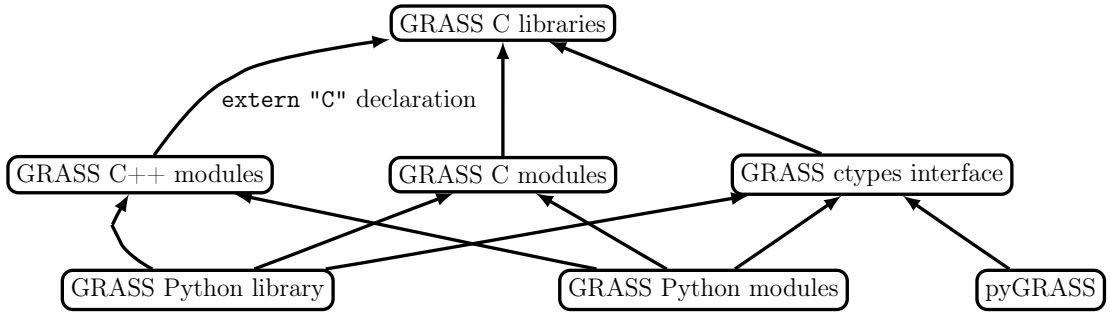


Figure 3.1: Ways to use GRASS libraries and modules; all nodes are part of the interface, so they are accessible to a programmer (note that ctypes interface and pyGRASS are in fact parts of GRASS Python library)

used QtCreator[5] which is an integrated development environment [6] (IDE) focused on C++ and Qt[7]. GRASS development requires some specific settings. I described this settings on GRASS wiki.[8]

Spyder is an IDE for Python language. I used it because it provides many features mainly connected to static source code analysis which is needed for dynamically typed language such as Python. To get everything working you should run Spyder form the GRASS environment (command: `spyder 2> /dev/null &`).

The main tool used for static source code analysis was Pylint[9]. It is integrated into Spyder but can be used also as a standalone application. There are some changes which need to be done to the standard Pylint settings in order to get correct results. I described these changes at GRASS wiki[10].

Profiling of C and C++ code is described in section 3.3.2 and Python code profiling is described in 3.3.3.

3.2 Language choice

At first, proper programming language was chosen for each module. The *i.edge* module implementing Canny edge filter does many computations, so C is the language of choice. It also corresponds to the default language choice for GRASS modules.

The *r.houghtransform* module implementing Hough transform requires a lot of data structures [43] which are, during the module runtime, transformed to different set of structures. C++ programming language is used for this because it provides both efficient memory management and expressive language to create desired structures. Furthermore, C++ language provides standard library, Standard template

[5] `http://qt-project.org/downloads`
[6] `http://en.wikipedia.org/wiki/Integrated_development_environment`
[7] `http://qt.digia.com/`
[8] `http://grasswiki.osgeo.org/wiki/Using_QtCreator_for_GRASS_C_development`
[9] `http://www.pylint.org/`
[10] `http://grasswiki.osgeo.org/wiki/Pylint_rc_file_for_GRASS`

library, which allows to take named advantages with minimal effort [49]. Note that C++98[11] was used to implement this module since C++98 currently ensures the highest portability and moreover, the code does not contain any constructs which would be significantly better to write in C++11[12].

All *i.farm.∗* modules use Python programming language (specifically, the Python versions 2.6 and 2.7). Some modules only call other GRASS modules directly while others access GRASS functionality through GRASS Python library or, when speed is an issue, through GRASS ctypes interface or pyGRASS interface. Python is used because it is light weight and easy-to-use when more programming techniques are combined (calling other modules, string manipulation, accessing database, computations, object oriented programming).

3.3 Speed improvements and optimization

All crucial modules were optimized in order to provide a user with a reasonably fast set of tools. Here an overview of improvements is presented to be an inspiration for further work. Some improvements may seem obvious to someone but it is worth mentioning them to increase also educational value of this work.

3.3.1 Improving *r.houghtransform* logic

An optional input — raster map with angles — was used to limit the number of pixels which need to be considered (see also section 2.1.2). The *r.houghtransform* module is at least two times faster when user provides edge angles.

3.3.2 Improving C++ code

The profiling tool used to identify the places which need an improvement was Valgrind [50]. More particularly, QtCreator Valgring (Callgrind tool [51]) integration was used. This integration provides an easy-to-use user interface which allows fast evaluation of results.

Profiling showed that the huge amount of time is spent by memory management i.e, in `malloc` or operator `new` calls. So I revised all places where new items are added to containers. and also places where there were repetitive definitions of variables.

In the latter case, there are two things which must be considered. The best practice is to define a variable in the most inner scope (function or loop). However, defining the variable before the loop ensures that the memory for this variable is allocated only once. For example, I moved definitions of two vectors from the most inner loop (listing 3.1 on the next page) to the function top level and the time spent in the function was decreased twice.

[11] ISO/IEC 14882:1998, `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=25845`

[12] ISO/IEC 14882:2011, `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=50372`

Listing 3.1: Moving variable definition out of inner loop ensures that the memory is allocated only once

```
// new place of definition
std::vector<int> indicesI(line_width);
std::vector<int> indicesJ(line_width);

// ...
while (!lineCoordinates.empty())

    // ...
    for (;; x += dx, y += dy)
    {
        // former place of definition
        std::vector<int> indicesI(line_width);
        std::vector<int> indicesJ(line_width);

       // ... working with vectors
```

In the case of adding items to container I have changed usages of `std::list` to `std::vector` which allows reservation (i.e. allocation) of memory without actual inserting of items. Consequently, when actually inserting an item no memory is allocated. This is more advantageous because memory allocation cost is similar for both large and small amounts of memory. So, it is worth to allocate memory in large amounts. Of course, you cannot apply it when you do not know the required amount of memory. Comparing this to plain C, this memory reservation is similar to working with plain array but thanks to C++ you have the advantage of high level and less error prone interface.

It is a fact that each function call has some cost. If we want to avoid this overhead for short, often called functions, we are using macros in C or their more safer C++ replacement — inline[13] functions. It may not be clear when to create a function, create an inline function or inline the code manually. If the profiling shows that some function is called many times from one place, it is worth testing the three mentioned possibilities to find the fastest one.[14]

3.3.3 Module call overhead issue

As mentioned above, there are two possibilities how to access GRASS functionality from Python. When profiling developed modules I realized that a lot of time is spent by calling other modules even though these called modules were doing only simple tasks. Although subprocess calls in Python are a complex topic[15], we can

[13] The inline keyword suggests compiler to replace function call by function code but compiler is not obliged to do it.

[14] Note that this may differ between various compliers.

[15] `http://bugs.python.org/issue11314`

Listing 3.2: Bash script which profiles a Python script and outputs PDF and PNG files with call graph

```
#!/bin/bash

# Usage:
# ./python-profile.sh your_python_script.py and its parameters

OUTFILE=$(basename $1 .py)_profile

python -m cProfile -o $OUTFILE.pstats $* \
&& gprof2dot -f pstats $OUTFILE.pstats > $OUTFILE.dot \
&& dot -Tpng -o $OUTFILE.png $OUTFILE.dot \
&& dot -Tpdf -o $OUTFILE.pdf $OUTFILE.dot
```

dare to say that the obvious explanation is the overhead of calling a subprocess.[16] However, since subprocess calls are very common for GRASS Python scripts, it is worth to compare subprocess (module) call to function call in GRASS Python script in a simple benchmark using a profiling tool (profiler) to see if this is really an issue.

In this work, Python cProfile tool was used for Python code profiling because this tool is recommended [52] and considered as fast [53]. The cProfile tool was used together with `gprof2dot`[17] and `dot` tools which visualize the profiling output as an call graph with percentages of time spent in particular functions (see listing 3.2). Example output is in the figure 3.2 on page 58.

For the module calling testing, I created two test modules. The first module (listing 3.3) loops over a given range. The second module (listing 3.4) calls the first module and also calls a function which does the same work as the first module. The module and the function are called ten times in the same loop to get more precise results. All modules and functions have a parameter — an integer — which specifies the range to loop over. This loop simulates some calculation.

The result of benchmark (table 3.1 on the following page) shows that for the large tasks the time for function call and module call is comparable. The reason is that the time needed for completing the task is much higher than the time needed for subprocess call. However, when the task is not so large the module call is incomparably slower than function call.

Although the conditions for this benchmark were not professional[18], the result corresponds with the practical experience with *i.farm.** modules development. Fortunately, the issue is not critical when module (not a function) is called only once or twice because the absolute clock time is still negligible. However, if some higher flexibility, i.e. calling a module for all vector map features, is needed, the overall time spent by subprocess calls is an issue.

[16] In fact, two subprocesses are called when calling a GRASS module in GRASS Python script because of *g.parser* call.

[17] `http://code.google.com/p/jrfonseca/wiki/Gprof2Dot`

[18] The test was performed on personal computer with Ubuntu. Results in the table comes from several profiler runs with different sets of other running applications.

Listing 3.3: The fist module for the subprocess call benchmark

```
#!/usr/bin/env python

#%module
#%description: Loops over range given by number and sums numbers
#%end
#%option
#% key: number
#% description: Loop maximum
#% type: integer
#% required: yes
#%end

import grass.script as grass

def do_work(number):
    a = 0
    for i in range(0, number):
        a += i
    return a

def main():
    options, flags = grass.parser()
    number = int(options['number'])
    do_work(number)

if __name__ == '__main__':
    main()
```

Table 3.1: The percentage of clock time spent in function and by module call for different input complexity of task (range is the size of range the module or function loops over)

range	module (%)	function (%)
10,000	94	2.6
100,000	72	21
1,000,000	60	39
10,000,000	53	46

Listing 3.4: The second (main) module for the subprocess call benchmark

```
#!/usr/bin/env python

#%module
#% description: Calls module and calls function
#%end
#%option
#% key: number
#% description: Loop maximum
#% type: integer
#% required: yes
#%end

import grass.script as grass

def call_module(number):
    # the name of first module is mt.a
    grass.run_command("mt.a", number=number)

def do_my_work(number):
    a = 0
    for i in range(0, number):
        a += i
    return a

def main():
    options, flags = grass.parser()

    number = int(options['number'])
    for i in range(0, 10):
        call_module(number)
        do_my_work(number)

if __name__ == '__main__':
    main()
```

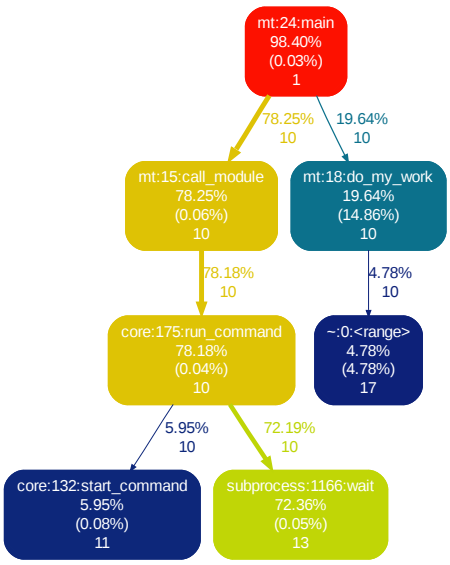


Figure 3.2: Example output of Python profiling; the output (intermediate `dot` file) was simplified because standard graph is very large; we can see the number of calls, percentage of (absolute not processor) time spent by calling a function and percentage of time spent in function itself (self-cost)

As a result, I have decided to use function calls instead of module calls in *i.farm.** modules whenever it is possible (see section 3.3.4), even though it is more complicated or less secure (in terms of data consistency or module stability).

Unfortunately, not all GRASS functionality is accessible by functions calls. In fact, a lot of functionality is accessible only by module calls.[19] In this work there are several examples of functionality which would be better to call as a function rather than a module because of the number of calls. These examples include computing profiles of a raster map and computing vector and raster statistics.

3.3.4 Improving Python code

Considering arguments stated in the section 3.3.3, all newly developed Python modules (referred as *i.farm.** modules) were revised in order to improve performance. Also some other changes which influence the performance were made and are listed here, too.

The improvement which addresses the module call issue is the usage of the new GRASS GIS Python interface called pyGRASS.[20] It was not possible to use it at the

[19] By counting lines in `*.c` files in `lib` directory and in `vector`, `raster` and `imagery` directories we can find out that approximately half of the GRASS source code is in the library.

[20] The current version of pyGRASS was developed during Google Summer of Code 2012, `https://gist.github.com/3864411`

beginning of this work but now the critical parts were rewritten using pyGRASS. The usage of the pyGRASS library together with other GRASS library calls (using ctypes) replaced many module calls (especially those which are called hundred of times like *g.region*). This change significantly improved the speed. As explained in section 3.3.3, it was possible to rewrite only functionality connected to use of the library.

Furthermore, the detection using profiles (described in the section 2.1.3) was replaced by detection based on building rectangle from line segments (described in the section 2.1.4). The implementation of the latter one is faster and there are also some other issues related to profiles such as complicated preparation of input data.

As described in the section 1.5, approximate positions of farms are known, so building detection runs only in a limited area (search window). One factor which can speed up or slow down the detection process is the size of the search window around approximate points. In some areas, we can notice that if the window is twice bigger, the computation time is increased by one half of the previous time. However, more important factor is the number of the detected rectangles (potential buildings) and this number highly depends on actual data as well as on the used algorithm. Therefore, the size of the search window can be set by a user which would like to set the search window anyway because of the better knowledge of how far the building can be from the approximate point.

After all changes the speed improvement was around 30% for one module. The resulting speed up for the whole detection was almost 50%. The absolute time, measured approximately on a personal computer, for 46 points with 70 buildings is around 20 minutes. This means less than 30 seconds for one point (searched rectangle was 200 by 200 meters). As an outlook, it must be said that the tools can be slowed down by additional features or different approaches. However, since GIS computations do not have to be real time, the speed is more than sufficient and enables comfortable testing and usage.

Chapter 4

Results

Results of this work consist of two parts. The first part is the implemented software which can be reused in other projects. The second part is the application of implemented software to farm detection in Northern Italy.

4.1 Implemented software

All implemented software is available as GRASS modules, particularly GRASS 7 modules. In this section, modules are divided into two groups. The first group contains modules which are general and can be used for any building detection algorithm. The second group contains modules which are specialized for farm building detection.

4.1.1 General modules

Here is the list and descriptions of modules which are general and intended to be used by other building detection algorithms. These modules were published in the official GRASS Addons[1] repository. The installation of GRASS Addons is described in the appendix A.

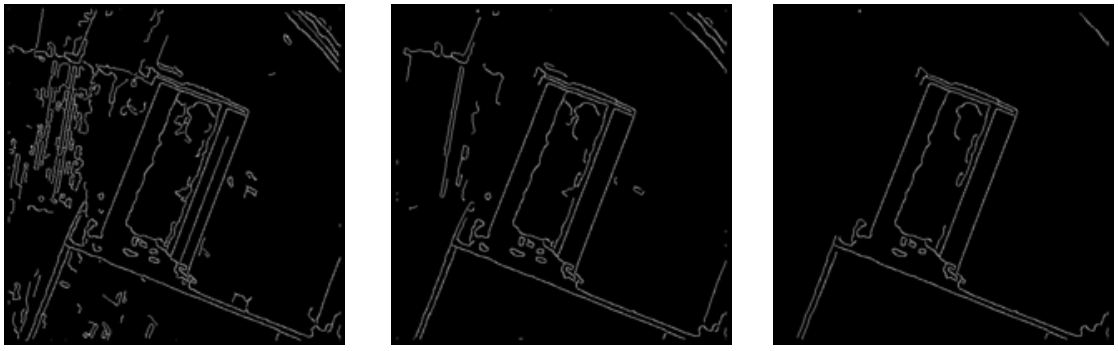
i.edge

The Canny edge algorithm (described in 2.1.1) is implemented in a new GRASS GIS module *i.edge*. It uses the Canny algorithm and it is based on [54].

By changing parameters of the module one can easily achieve different levels of detail. There are 3 parameters which affect the result — σ value and two threshold values (lower lT and higher hT). Their effect is visible from figure 4.1. It is generally recommended [40] to use lT and hT threshold values which have its ratio (computed as hT/lT) between 2 and 3.

The *i.edge* module is very general (see figure 4.2 on page 63) and so it can be used to accomplish many different tasks.

[1] `http://grass.osgeo.org/download/addons/`



(a) $\sigma = 2,\ lT = 3,\ hT = 10$ (b) $\sigma = 2.5,\ lT = 3,\ hT = 10$ (c) $\sigma = 2.5,\ lT = 3,\ hT = 15$

Figure 4.1: Result of *i.edge* with different initial values

r.houghtransform

The Hough transform (section 2.1.2) was implemented in the *r.houghtransform* module.

The main purpose is to detect line segments, so the output of the module is a vector map which contains line segments found in the input raster edge map. The level of details is controlled by several parameters.

The continuous straight series of pixels are interpreted as a part of a line when they are not so scattered (line width is limited). The method tolerates small and not too often repeated gaps. As a result, the *r.houghtransform* module can produce more than one line segment for one detected line when the gap is too long. The approximate number of all resulting lines can be specified by the user. All the limits mentioned above (minimum length of line segment, size of the gaps) can also be controlled by the user. The example result of the Hough transform can be seen in figure 4.3 on the facing page.

The optional output raster map (direction of edges) of module *i.edge* can serve as an input to *r.houghtransform* module. This reduces significantly the time needed for the computation without negative effect on the result.

The optional output of *r.houghtransform* module is the image transformed into the Hough space, i.e. the original output of Hough transform. It can be used for further processing and analysis if desired. The image is outputted as a map at coordinates (0,0). One image pixel is represented as one cell but the image does not have geographical meaning (remark Hough space). The color table of this image is set to gray scale with the black representing a zero. Note that the number of curves in this image is reduced when you provide angle map as an optional input. So, if you want to get nicely looking image, you shall not provide angle map to *r.houghtransform* module.

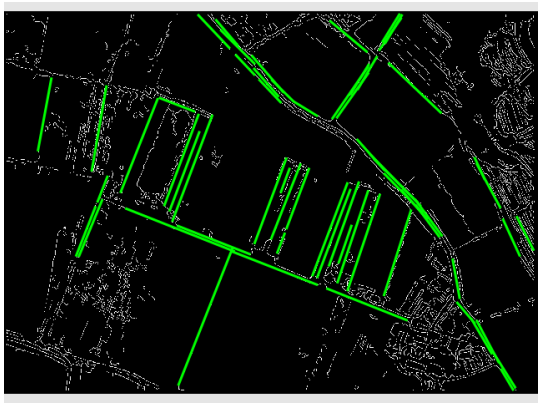
v.class.mlpy

The new module *v.class.mlpy* is a tool for supervised vector classification. It is built on top of the Python mlpy library [46] (the current version is 3.5).

(a) Original photograph

(b) Canny edge detector applied

Figure 4.2: The module *i.edge* applied on intensity channel of an photograph containing various types of edges. Note how hair on the face which is hardly visible at the original photograph was detected as edges. Also note the trees and electricity pole in the background which shows what detector evaluated as edge and what not. The used parameters follow: $\sigma = 1,\ lT = 3,\ hT = 5$



(a) Line segments overlaid on the Canny image

(b) Line segments overlaid on the original image

Figure 4.3: Line segments detected by the Hough transform

The classification is based on attribute values. The geometry is not taken into account so the module does not depend on the feature types used in the map. The classification is supervised so the training dataset is always required. The attribute table of training map (dataset) has to contain a column with the class. Required type of class column is integer. Expected type of other columns is double or integer.

This module requires the user to have mlpy library installed. However, this is not an issue because mlpy library is free and open source and can be quickly downloaded and installed.[2] Furthermore, library is available for all platforms supported by GRASS.

This module was at some development stage used but now it is not because of problems with obtaining sufficient training data. However, performance and general experience is good so it may be used by our project again when desired.

Here, I will present a basic test (listing 4.1 on the next page) which proofs whether module basic functionality works. The test combines the usage of sample data provided by GRASS project (North Carolina dataset)[3] and randomly generated data.

Two sets of random points were generated containing 1,000 and 10,000 points. Then, an attribute table was created for both maps and attributes were derived from digital values of raster maps (Landsat images) at points locations. This means that attribute table columns are input to the classification. The smaller dataset is used as training dataset. Classes were taken from the raster map which is a part of sample dataset as an example result of some former classification. The number of classes in training dataset is 6.[4] The test is not intended to measure success rate of the mlpy classifier, however success rate is high enough to see from the result (figure 4.4 on page 66) that the module works properly.

4.1.2 Modules specific for farm detection

Besides truly general modules available from GRASS Addons (listed in 4.1.1) there are newly developed modules which are focused on the farm detection use case. These modules are accessible from FEM[5] GRASS Addons repository as described in the appendix A.

i.farm.mask.classification

This module encapsulates raster classification based on signature file and given raster maps. The output is a map which can be used as a mask. The mask is created from a selection of classes specified by a parameter.

[2] `http://mlpy.sourceforge.net/docs/3.5/`

[3] `http://grass.osgeo.org/download/sample-data/`

[4] The number of classes in original map is 7 but only a few pixels are in class 7.

[5] `http://www.iasma.it/`

Listing 4.1: The testing script which shows how *v.class.mlpy* works. Is is intended to run in the GRASS North Carolina sample dataset. It generates random points and uses raster maps (landsat images) to fill their attribute tables. Then, the class is added to the training dataset. At the end, classification is performed on random points using training dataset.

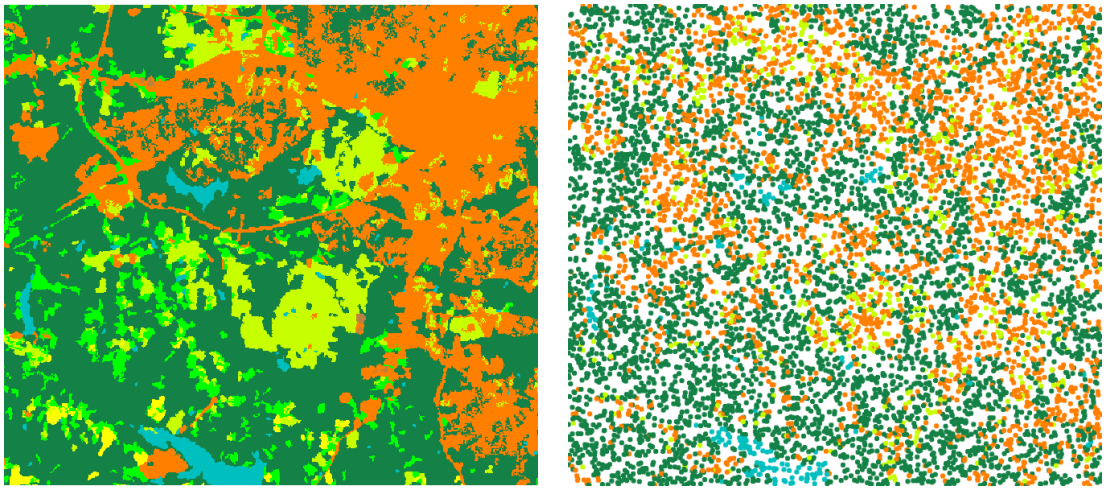
```
# generate random points used as an input
v.random output=points_unknown n=10000
v.db.addtable map=points_unknown

# generate random points used as training dataset
v.random output=points_known n=1000
v.db.addtable map=points_known

# fill attribute tables
MAPS=$(g.mlist type=rast pattern="lsat*" exclude="*87*" \
       mapset=PERMANENT sep=" ")
let NUM=0
for MAP in $MAPS
do
  let NUM++
  v.db.addcolumn map=points_unknown layer=1 \
                 columns="map_$NUM integer"
  v.db.addcolumn map=points_known layer=1 \
                 columns="map_$NUM integer"
  v.what.rast map=points_unknown layer=1 raster=$MAP \
                 column=map_$NUM
  v.what.rast map=points_known layer=1 raster=$MAP \
                 column=map_$NUM
done

# fill the class column
# with correct values for training dataset
v.db.addcolumn map=points_known layer=1 \
               columns="landclass integer"
v.what.rast map=points_known layer=1 raster=landclass96 \
            column=landclass

# do the classification
v.class.mlpy input=points_unknown training=points_known \
             class_column=landclass
```



(a) raster map `landclass96` from the GRASS GIS North Carolina sample dataset

(b) random points classified according to training dataset and various raster maps

Figure 4.4: Maps from mlpy classifier test; (a) is a used for creating training dataset and (b) is the resulting classified dataset which were classified using values from various raster maps by *v.class.mlpy*

i.farm.rectangles.stats

This module loops over the list of rectangles or other polygons in a map. For each feature (based on category), a set of statistical values is computed. The advantage of this module over other existing GRASS modules with similar functionality (*v.rast.stats*, *v.univar*, *v.what.rast*) is that it computes valid raster statistics even for rectangles which cannot be areas because they are overlapping with other rectangles, thus their topology is invalid. The result is stored in the attribute table.

For building detection algorithms, it is very common that potential building polygons (building hypothesis) overlaps each other. The raster statistics have to be obtained before removing these overlapping features. For this reason, functionality provided by this module is crucial for many detection methods including also LiDAR based detections using raster data available (see section 1.4).

This module is general but currently it supports only one way of representing overlapping polygons. This should be changed before moving this module into GRASS Addons.

i.farm.detect.rectangles

This module implements algorithm presented in the section 2.1.4. A rectangle is composed from 2 or more line segments which are determined from several runs of *i.edge* and *r.houghtransform* modules. A mask map is used to filter lines which would lead to obvious false detection.

i.farm.detect.buildings

This is the main detection module which encapsulates almost all other modules. It contains almost all assumptions made about buildings in farm detection use case. Therefore this module is the most specific module from *i.farm.** modules. It currently supports only detection of light roof buildings because these are definitively the most frequent ones (see section 1.5).

The input for this module consists of several maps which should overlap with the vector map containing approximate farm positions. The output is a set of detected buildings. Attributes coming from detection are preserved, so it is possible to do further processing or algorithm evaluation using these values.

i.farm.rast.window

The input is a set of raster maps and a region (specified as a coordinates pair and size of a window). The module cuts the given region from the given set of maps. This is particularly useful when the input for some algorithm is a set of points (places) where to do an analysis and a set of map which cover large area but are stored as separate maps.

i.farm.rectangles.compare

When testing the implemented tools and tool chains we need to compare rectangles in two maps. This module loops over rectangles in both given maps and reports how many rectangles are missing in each map. More particularly, it computes statistics described in the section 2.3.

i.farm.duplicates

This module filters duplicate rectangles contained in the map. It tries to remove duplicated rectangles using their attribute values. When module is not able to decide, both overlapping rectangles are preserved. It is also possible to run module in mode when only very similar (in terms of position and shape) rectangles are considered as duplicates.

i.farm.mask.redfields

This module aims at detection of red fields (section 2.2.1). The red fields are determined using hue and saturation values. The advantage of this module is that it detects only areas which are large enough to be fields using *r.reclass.area* and *r.grow* modules.

4.2 Application to Northern Italy images

The implemented modules were used for farm building detection on Northern Italy orthophotos (described in the section 1.5). This section describes how the whole

algorithm works and how it is successful. The user guide describing how to use new modules in the way required by this algorithm is described in the appendix C.

4.2.1 Detection procedure

The input for the detection is the set of points, representing approximate farm positions, and RGB images (section 1.5). The detection method is based on the elongated shape of the farm buildings and on the color and homogeneity of their roofs.

The process is run stepwise in the areas specified by the approximate points. The detection is performed in the following way:

1. As input, the intensity channel extracted by HIS transform from RGB orthophoto is used.
2. An edge map is created by the *i.edge* module.
3. The *r.houghtransform* module extracts line segments from the edge map.
4. Rectangles representing potential farm building are combined from line segments obtained from several *i.edge* and *r.houghtransform* runs.
5. For each area limited by the rectangle basic statistical values from underlying raster layers are computed.
6. These statistics are then used to determine possible farm buildings.
7. Duplicate matches are removed.

Note that some parts were simplified in order to better show the basic steps.

The described steps are graphically expressed at the figures 4.5 on the next page and 4.6 on page 70. Figure symbology is partially derived from symbology used by GRASS GIS Modeler.[6] Legend to figures is blue — input and output, red — other data, green — modules and functions. Modules or functions are represented as rounded rectangles; data by ellipses (ovals).

According to the searched building specification (section 1.5), the algorithm is focused on elongated farm buildings. Consequently, while some smaller squared buildings are not detected, farm buildings are detected successfully (figure 4.7 on page 70). Moreover, the specification allows to use assumptions about digital values in the area of a building, so the algorithm is specialized only to light gray roofs. The other buildings e.g. green buildings are not detected (figure 4.7 on page 70).

4.2.2 Results of detection

The results of the detection were compared to manually digitized data containing 75 buildings. The success was measured by using criteria described in the section 2.3. The detection percentage was 70 % and branch factor was 24 %.

[6] `http://grass.osgeo.org/grass70/manuals/g.gui.gmodeler.html`

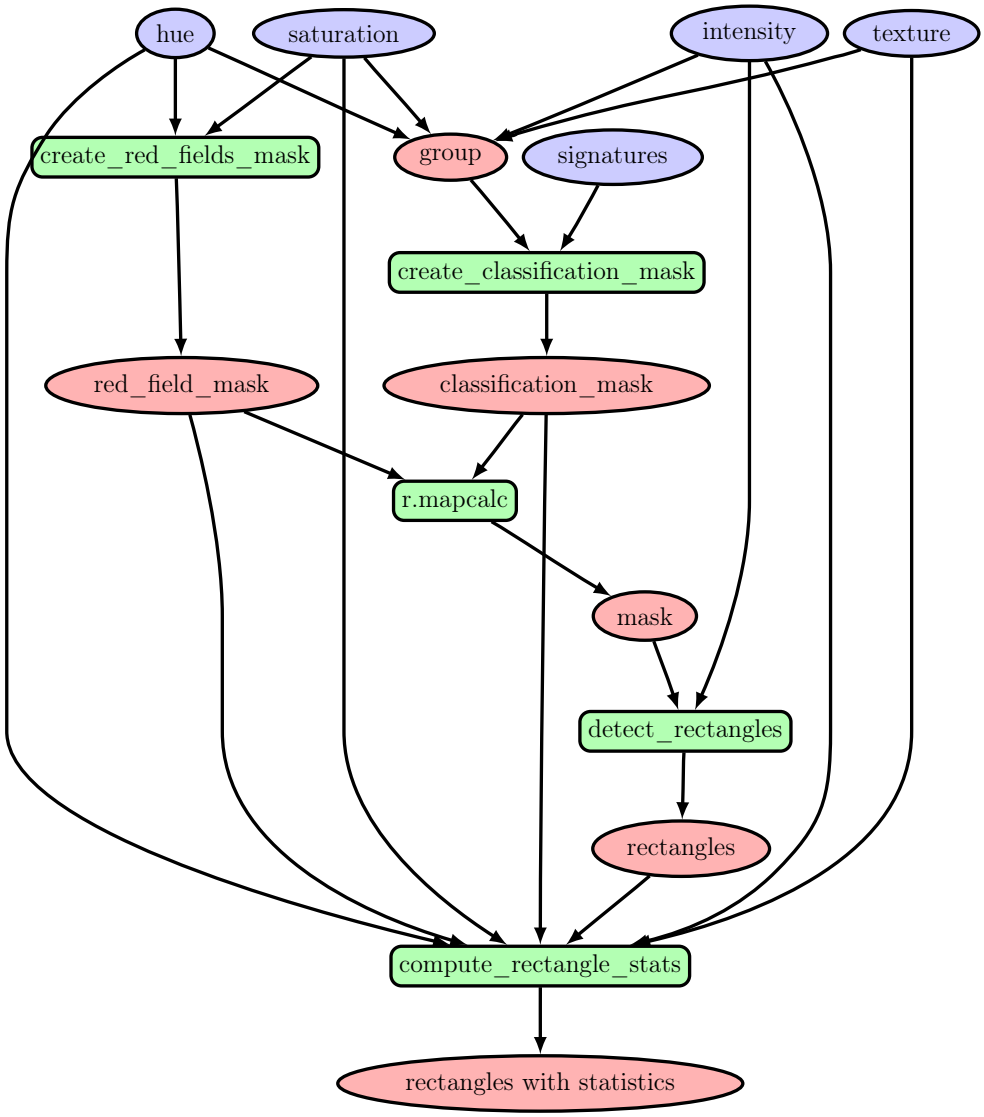


Figure 4.5: Building detection overview — part I. The first part of building detection when all raster maps are processed, rectangles are detected and vector map with attributes derived from raster maps is created (blue — input and output, red — other data, green — modules and functions)

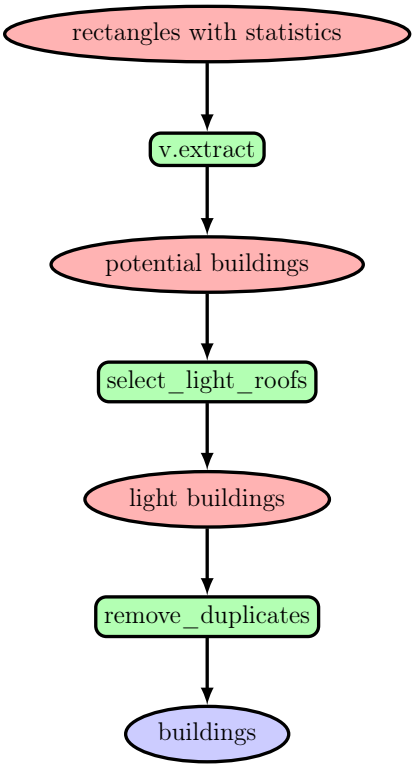


Figure 4.6: Building detection overview — part II. The second part of building detection when rectangles representing buildings are extracted from vector map. In the second phase duplicates are removed. Extraction and removing of duplicates are based on attributes (blue — input and output, red — other data, green — modules and functions)

(a) farms in the village (b) farms with light gray and green roofs

Figure 4.7: Detected farm buildings; the figure (a) shows successful detection in the area with other building; the figure (b) shows only partial success when roofs with color similar to the background and roof with dark spots were not detected

More particularly, the tests were done on an area of 80 square kilometers with 46 approximate points. Input data is an RGB orthophoto with defined resolution 0.5 m. The actual resolution and aerial image properties were not specified, however it seems that the illumination and illumination angle are consistent and the actual image resolution is more than 0.5 m. The number of manually identified buildings was 75 and the algorithm correctly detected 53 of them; another 17 detected buildings were false positives. It must be noted that the part of the algorithm detecting rectangles detects almost all buildings i.e., more than 90 %, however subsequent filtering which should remove false positives removes also true positives. Consequently, not all detected rectangles which represent buildings are identified as buildings.

Conclusion

The variability of landscape, aerial image qualities and detected objects often lead to the creation of specialized tools. Moreover, there can be additional data available which improves or possibly simplifies the detection. This work presents tools developed for farm building detection on the northern Italy orthophotos. These buildings are characterized by an elongated shape, so the presented detection procedure is focused accordingly. Furthermore, approximate positions of groups of farm buildings are available. Thus, the detection can benefit from this data. In the opposite case, when the algorithm would iterate over the whole area, the performance would suffer from the low density of farm buildings. Despite the necessary specialization of the detection tools, the part of the presented tools is general and available for further usage.

This work presented three newly implemented GRASS modules which are now available in the GRASS Addons repository, namely *i.edge*, *r.houghtransform* and *v.class.mlpy*. The *i.edge* module implementing Canny edge detector provides an easy-to-use reliable general edge detector. The *r.houghtransform* module has highly customizable interface to powerful line segment extraction algorithm whose main pillar is the Hough transform. The *v.class.mlpy* module which allows to classify vector features according to their attributes is based on mlpy library. This module shows how much a free and open source projects can gain by delegating some work to other projects. This corresponds to the free software development principles as well as to the well know reduce, reuse, recycle principle.

Furthermore, this work presents modules which were developed for the farm building detection on the northern Italy orthophotos (*i.farm.** modules). The set of modules make use of the general modules listed above (*i.edge* and *r.houghtransform*). The detection results are sufficient considering the available input data and its quality. Additionally, the detection procedure is relatively fast thanks to the optimization and the fact that approximate positions are known in the study area. The implemented modules are not part of the GRASS GIS project but are available in a specialized repository.

Further work on farm detection use case may include extending building detection to red, green and dark roofs. Considering issues encountered during experiments with red and green roofs, the improvement of vegetation detection have to precede

in order not to interfere with the detection of named roofs. Furthermore, the usage of segmentation algorithms should be reconsidered because of the latest advances in the development (modules *i.segment* and *i.segment.xl*). Moreover, some of the implemented modules can be generalized to serve to a wider range of users. Currently, a specialized lightweight GUI tool is under development. The purpose of this tool is to enable easy post-classification and quick corrections of detected buildings.

In conclusion, the presented set of tools is based on GRASS GIS and also partially incorporated into this free and open source software project. For the further use of the implemented software, it is crucial that not only the newly implemented software but also the underlying GIS environment is available. Moreover, the free and open source solution ensures that the presented work is highly reproducible and that the newly implemented tools can be reviewed and reused.

Bibliography

[1] A. Kratochvílová and V. Petráš. *Detekce obdélníků v obraze pomocí Houghovy transformace a použití této metody pro hledání budov farem na ortofotech severní Itálie*. Tech. rep. České vysoké učení technické v Praze, Fakulta stavební, 2012.

[2] J. Demel. *Grafy a jejich aplikace*. Academia, 2002.

[3] Inc. Free Software Foundation. *The Free Software Definition*. [Online; accessed 22-September-2012]. 2012. URL: `http://www.gnu.org/philosophy/free-sw.html`.

[4] R. Stallman et al. "The GNU manifesto". In: *Dr. Dobb's Journal of Software Tools* 10.3 (1985), pp. 30–35.

[5] R. Stallman. *Why Open Source misses the point of Free Software*. [Online, accessed 22-September-2012]. 2010. URL: `http://www.gnu.org/philosophy/open-source-misses-the-point.html`.

[6] K. Fogel. *Producing open source software: How to run a successful free software project*. O'Reilly Media, Inc., 2005.

[7] K.J. Boudreau and K.R. Lakhani. "How to manage outside innovation". In: *MIT Sloan management review* 50.4 (2009), pp. 69–76.

[8] F. Potortì. "Free software and research". In: *proceedings of the International Conference on Open Source Systems (OSS)*. 2005, pp. 270–271.

[9] D. Rocchini and M. Neteler. "Let the four freedoms paradigm apply to ecology". In: *Trends in Ecology and Evolution* (2012).

[10] M. Neteler et al. "GRASS GIS: A multi-purpose open source GIS". In: *Environmental Modelling & Software* 31.0 (2012), pp. 124–130. ISSN: 1364-8152.

[11] M. Neteler and H. Mitasova. *Open source GIS: a GRASS GIS approach*. Vol. 773. Springer, New York, 2008.

[12] P. Souček and J. Formánek. "Data spravovaná resortem ČÚZK jsou stále přístupnější". In: *GIS Ostrava 2012 - Současné výzvy geoinformatiky*. 2012.

[13] M. Haklay and P. Weber. "Openstreetmap: User-generated street maps". In: *Pervasive Computing, IEEE* 7.4 (2008), pp. 12–18.

[14] N. Shorter and T. Kasparis. "Automatic vegetation identification and building detection from a single nadir aerial image". In: *Remote Sensing* 1.4 (2009), pp. 731–757.

[15] J. Rosen and J. Shamir. "Circular harmonic phase filters for efficient rotation-invariant pattern recognition". In: *Applied optics* 27.14 (1988), pp. 2895–2899.

[16] J. Rosen and J. Shamir. "Scale invariant pattern recognition with logarithmic radial harmonic filters". In: *Applied optics* 28.2 (1989), pp. 240–244.

[17] H. Kim and S. de Araújo. "Grayscale template-matching invariant to rotation, scale, translation, brightness and contrast". In: *Advances in Image and Video Technology* (2007), pp. 100–113.

[18] K. Karantzalos and N. Paragios. "Automatic model-based building detection from single panchromatic high resolution images". In: *ISPRS Archives* (2008).

[19] D. Rainsford and W. Mackaness. "Template matching in support of generalisation of rural buildings". In: *Advances in Spatial Data Handling. Proceedings 10th International Symposium on Spatial Data Handling, Berlin Heidelberg: Springer*. 2002, pp. 137–151.

[20] R. Mathieu, J. Aryal, et al. "Object-based classification of Ikonos imagery for mapping large-scale vegetation communities in urban areas". In: *Sensors* 7.11 (2007), pp. 2860–2880.

[21] V. Verma, R. Kumar, and S. Hsu. "3D building detection and modeling from aerial LIDAR data". In: *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*. Vol. 2. IEEE. 2006, pp. 2213–2220.

[22] U.C. Benz et al. "Multi-resolution, object-oriented fuzzy analysis of remote sensing data for GIS-ready information". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 58.3 (2004), pp. 239–258.

[23] A. Fischer et al. "Extracting buildings from aerial images using hierarchical aggregation in 2D and 3D". In: *Computer Vision and Image Understanding* 72.2 (1998), pp. 185–203.

[24] J.A. Shufelt. "Exploiting photogrammetric methods for building extraction in aerial images". In: *International Archives of Photogrammetry and Remote Sensing* 31 (1996), B6.

[25] F. Rottensteiner and C. Briese. "A new method for building extraction in urban areas from high-resolution LIDAR data". In: *International Archives of Photogrammetry Remote Sensing and Spatial Information Sciences* 34.3/A (2002), pp. 295–301.

[26] T. Vogtle and E. Steinle. "3D modelling of buildings using laser scanning and spectral information". In: *International Archives of Photogrammetry and Remote Sensing* 33.B3/2; PART 3 (2000), pp. 927–934.

[27] F. Rottensteiner et al. "Building detection using LIDAR data and multispectral images". In: *Digital Image Computing: Techniques and Applications*. Vol. 2. CSIRO. 2003, pp. 673–682.

[28] J. Secord and A. Zakhor. "Tree detection in urban regions using aerial lidar and image data". In: *Geoscience and Remote Sensing Letters, IEEE* 4.2 (2007), pp. 196–200.

[29] M. Rutzinger et al. "Object-based building detection based on airborne laser scanning data within GRASS GIS environment". In: *Proceedings of UDMS*. Vol. 2006. 2006, 25th.

[30] M. Rutzinger et al. "Object-based analysis of airborne laser scanning data for natural hazard purposes using open source components". In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 36.4/C42 (2006).

[31] G. Pratx. *Building Detection*. Tech. rep. [Online; accessed 05-July-2012]. LIAMA, 2004. URL: `http://donut.99.free.fr/En-vrac/liama/buildingdetection.pdf`.

[32] T. Kim and J.P. Muller. "Development of a graph-based approach for building detection". In: *Image and Vision Computing* 17.1 (1999), pp. 3–14.

[33] B. Jähne. "Practical handbook on image processing for scientific and technical applications". In: (2004).

[34] J. Canny. "A Computational Approach to Edge Detection". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 8.6 (June 1986), pp. 679–698. ISSN: 0162-8828.

[35] B. Sirmacek and C. Unsalan. "Building detection from aerial images using invariant color features and shadow information". In: *Computer and Information Sciences, 2008. ISCIS'08. 23rd International Symposium on*. IEEE. 2008, pp. 1–5.

[36] Y. LI and H. WU. "Adaptive building edge detection by combining LiDAR data and aerial images". In: *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 37 (2008), pp. 197–202.

[37] L.R. Liang and C.G. Looney. "Competitive fuzzy edge detection". In: *Applied soft computing* 3.2 (2003), pp. 123–137.

[38] L. Chen et al. "Building reconstruction from LIDAR data and aerial imagery". In: *International Geoscience and Remote Sensing Symposium*. Vol. 4. 2005, p. 2846.

[39] J.C. Russ. *The image processing handbook*. CRC, 2011.

[40] M. Sonka, V. Hlavac, and R. Boyle. "Image processing, analysis, and machine vision". In: (1999).

[41] P. Zimmermann. "A new framework for automatic building detection analysing multiple cue data". In: *International Archives of Photogrammetry and Remote Sensing* 33.B3/2; PART 3 (2000), pp. 1063–1070.

[42] C. R. Jung and R. Schramm. "Rectangle Detection based on a Windowed Hough Transform". In: *Proceedings of the Computer Graphics and Image Processing, XVII Brazilian Symposium*. SIBGRAPI '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 113–120. ISBN: 0-7695-2227-0.

[43] M. Fiala. "Identify and Remove Hough Transform Method". In: *Proc. Vision Interface*. 2003, pp. 184–187. URL: `http://www.cipprs.org/papers/VI/VI2003/papers/S5/S5_fiala_28.pdf`.

[44] C. Galambos, J. Kittler, and J. Matas. "Using gradient information to enhance the progressive probabilistic Hough transform". In: *Pattern Recognition, 2000. Proceedings. 15th International Conference on*. Vol. 3. IEEE. 2000, pp. 560–563.

[45] C. Lin and R. Nevatia. "Building Detection and Description from a Single Intensity Image." In: *Computer Vision and Image Understanding* 72.2 (Sept. 11, 1998), pp. 101–121.

[46] D. Albanese et al. *mlpy: Machine Learning Python*. 2012. eprint: `1202.6548`.

[47] S. Müller and D. Zaum. "Robust Building Detection in Aerial Images". In: *Proc. Joint Workshop of ISPRS and DAGM: CMRT05*. 3. 2005, pp. 143–148.

[48] C. Heipke et al. "Evaluation of automatic road extraction". In: *International Archives of Photogrammetry and Remote Sensing* 32.3 SECT 4W2 (1997), pp. 151–160.

[49] U. Breymann. *Designing Components with the C++ STL*. Addison-Wesley, 1998.

[50] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan Notices* 42.6 (2007), pp. 89–100.

[51] J. Weidendorfer, M. Kowarschik, and C. Trinitis. "A tool suite for simulation based analysis of memory access behavior". In: *Computational Science-ICCS 2004* (2004), pp. 440–447.

[52] Python Software Foundation. *The Python Profilers*. [Online, accessed 06-December-2012]. 2012. URL: `http://docs.python.org/2/library/profile.html`.

[53] Ø. I. Øvergaard. "Controlling overhead in large Python programs". MA thesis. University of Oslo, 2012.

[54] Tom Gibara. *Canny Edge Detector Implementation*. [Online; accessed 20-June-2012]. 2010. URL: `http://www.tomgibara.com/computer-vision/canny-edge-detector`.

Appendix A

Installation guide

A.1 Required components

1. GRASS 7
2. *i.edge* and *r.houghtransform*
3. farm detection tools

A.2 Installation

A.2.1 GRASS 7

Please follow general compile and install instructions at GRASS Wiki or use binaries available from GRASS web site. Alternatively, you can use OSGeo4W Installer for MS Windows or packages provided by your GNU/Linux distribution (if GRASS 7 is provided).

Note that the latest GRASS 7 SVN version (daily snapshot) is required since the new pyGRASS python library is used.

A.2.2 Modules from Addons

Extensions from GRASS Addons can be installed through dialog which is in the menu:

Settings → Install extension from add-ons

Repository should be set to:

`http://svn.osgeo.org/grass/grass-addons/grass7`

This value is the default value so you probably don't have to set it. Then press Fetch button and then in List of extensions select

raster → r.houghtransform

end press Install button.

Installation progress and output can be seen in command console. Compiling gives lot of warnings since compiler is set to verbose level so no need to worry about it.

For installing the second module just select
imagery → i.edge

and press Install button again.

Alternatively, you can follow general addons installation instructions. If you have special needs, you can refer to *g.extension* manual.

A.2.3 Farm detection tools

If you have access to FEM GRASS Addons you can install farm detection modules from there. Just follow instructions in section A.2.2 and set the repository to:

`http://svn.fem-environment.eu/svn/grass_addons/grass7/`

Then press Fetch button and follow the instructions in section A.2.2. You need to install extension:
imagery → farmdetection

Alternatively, if you obtained a tar archive (`farmdetection.tar.gz`) with all sources, extract the archive in `imagery` directory in GRASS source code:

```
cd imagery
tar xvf farmdetection.tar.gz
```

Then enter the `farmdetection` directory and run `make` to compile and install modules.

```
cd farmdetection
make
```

A.3 Testing of the installation

A.3.1 Testing if pyGRASS works

1. go to the Python shell tab
2. run following Python commands

```
import pygrass
pygrass.raster.Region()
```

3. output should be similar to:

```
projection: 1 (UTM)
zone:       32
datum:      wgs84
ellipsoid:  wgs84
north:      5040330.33677
south:      5039597.35061
west:       668577.877011
east:       669600.857697
nsres:      0.49999056
ewres:      0.49999056
rows:       1466
cols:       2046
cells:      2999436
```

A.3.2 Testing if *i.edge* and *r.houghtransform* work

1. set computation region to small area e.g., 200 by 200 cells
2. run *r.houghtransform* on one image from RGB channels or on intensity image form HIS channels (default parameters should be enough to get meaningful results)
3. run *r.houghtransform* on result of *i.edge* (to get meaningful results changing of parameters may be needed but to just test the module default parameters should be enough)

Appendix B

Input data import

This text is intended to be a guide to import data to GRASS specialized for Raster-Vet project. Described files and file formats are currently used in this project. In case you want to use this guide generally, you can ignore some sections.

This guide assumes that you already have a GRASS location (projection UTM zone 32).

B.1 Approximate points import

Points associated with one farm are referred as approximate points. Geographic coordinates and some less important attributes of this points are in the shapefile while some more attributes are in CSV file. Import of both and their combination are covered here.

B.1.1 Shapefile with points

Transform data from Gauss-Boaga to UTM zone 32 (EPSG:32632[1]) in command line using GDAL/OGR:

```
ogr2ogr -t_srs EPSG:32632 UnitaEpidemiologiche_utm32 \
        1_UnitaEpidemio_SHP/UnitaEpidemiologiche.shp
```

Import data to GRASS through main menu in GRASS Layer Manager:
File → Import vector → Import common vector formats

B.1.2 CSV file with attributes

For successful import to SQLite it is necessary to simplify CSV file. Replace all commas between two double quotes (`","`) by separate comma (`,`), delete double quotes `"` at the line begin and end and delete header.

`","`	`,`
`"` (at the line begin and end)	nothing
header	nothing

[1] urn:ogc:def:crs:EPSG::32632

If you are sure about the CSV file you have, you can more simply just remove all occurrences of double quote character (") and header. You can use your favorite text editor or you can use this command:

```
cat input.csv | tr -d \" > output.csv
```

Now you need to to import CSV into SQLite database. The best option is to use Sqliteman[2] as a tool and to import CSV into database file — a GRASS internal SQLite database. The GRASS SQLite database of your mapset is in the directory:

```
.../locationname/mapsetname/sqlite
```

Before manual modification of the database file it is good idea to make a backup copy of it, e.g. by command:

```
cp sqlite.db sqlite.db.backup
```

In the database create table using GUI or using a SQL command (listing B.1 on the next page). Then you can actually import data to database from Sqliteman main menu:
Database → Import table data → Table import to, File to import → Sqlite .import → choose Comma → check Preview → OK

B.1.3 Combining data coordinates and attributes

Once imported, you can create new table (listing B.2 on the facing page) which combines imported data with attributes (and categories) which belongs to the map with approximate points. For the import and creating new database you can also use SQLite command line.

After import it is necessary to change the table associated to map layer. In other words the table for layer one needs to be changed to `epidemiologic_points` table. Alternatively it is possible to create new layer (layer two) in map, copy categories from layer one and associate the new table with this new layer. This can be changed in GRASS Attribute Manager:
Manage layers → Modify layer → Table

B.2 Importing rasters in GeoTIFF format

Firstly, rasters in GeoTIFF format need to be transformed to UTM 32. Note that the commonly used definition of Gauss-Boaga projection is not sufficient. Please, use proper `.prj` file instead.

You can type command to command line, however more convenient way is to create a shell script. If you created script similar to script in listing B.3 on page 86, you can convert all files in one directory by commands similar to these:

```
mkdir geotiff_utm32
./convert_geotiffs_to_utm32.sh
```

[2] http://sqliteman.com/

Listing B.1: SQL create statement for temporary table with attributes

```
CREATE TABLE avi_tuttelespecie_short_csv (
    "field_1" CHARACTER,
    "INSCOD317" CHARACTER,
    "GSATTRBRO" CHARACTER,
    "GSATTRRIP" CHARACTER,
    "GSATTROFP" CHARACTER,
    "GSATTRODA" CHARACTER,
    "GSATTRODB" CHARACTER,
    "GSATTRODG" CHARACTER,
    "GSATTRODT" CHARACTER,
    "GSTAVICPOT" INTEGER,
    "TSATTRTPRIP" CHARACTER,
    "TSATTRTPCAR" CHARACTER,
    "TSTAVICPOT" INTEGER,
    "ASAVIANA" CHARACTER,
    "ASAVICOT" CHARACTER,
    "ASAVIFAG" CHARACTER,
    "ASAVIFAR" CHARACTER,
    "ASAVIOCH" CHARACTER,
    "ASAVIPER" CHARACTER,
    "ASAVIPIC" CHARACTER,
    "ASAVIQUA" CHARACTER,
    "ASAVIRAT" CHARACTER,
    "ASAVISTA" CHARACTER,
    "ASTAVICPOT" INTEGER,
    "AUTOCONSUMO" CHARACTER
)
```

Listing B.2: SQL create statement to create new combined table

```
CREATE TABLE "epidemiologic_points"
    AS SELECT cat, CODICE317, CODGEOCODI, TIPOVALIDA,
              GSTAVICPOT, TSTAVICPOT, ASTAVICPOT, AUTOCONSUMO
        FROM avi_tuttelespecie_short_csv
        JOIN unitaepidemiologiche ON INSCOD317 = CODICE317
```

Listing B.3: Shell script to transform GeoTIFFs (directories and file names are hard coded)

```
#!/bin/bash

for FILE in ./geotiff/*.tif
do
    gdalwarp -s_srs ~/prj_files/gaussboaga1.prj \
    -t_srs epsg:32632 -tr 1.0 1.0 -r bilinear \
    $FILE geotiff_utm32/$(basename $FILE)
done
```

For actual import you can use either GRASS GUI or GRASS command line. In GUI you need to do the following:
File → Import raster data → Common formats import → select Directory → choose type GeoTIFF → Browse → unselect Add imported layers into layer tree → Import

Than it is possible to create RGB composite map by the module *r.composite*. Alternatively, you can use a script (listing B.4 on the facing page) which will do both the import and the creation of composite maps. Import will create many maps. If it is more convenient for you, it is possible to create one large map (listing B.5 on the next page). Note that this map can be a performance issue since it needs a lot of disk space.

B.3 Training data import

The import of training data is very similar to the import of point data (section B.1.1 on page 83) since both maps are in shapefile format. Again, data needs to be transformed and then imported to GRASS, for specific commands see listings B.6 on the next page.

B.4 Formal input files description

Legend for column names (attributes) in the file `AVI_TUTTELESPECIE_SHORT.csv` is in the table B.1. Other data or attributes are considered as self explaining or not important for this work.

B.5 Unexpected attribute combination

The file `AVI_TUTTELESPECIE_SHORT.csv` contains several records with an unexpected combination of attributes. These records have `AUTOCONSUMO='S'` and a number of animals higher than 1000.

These records can be obtained by the following SQL command:

Listing B.4: Shell script to import multiple GeoTIFFs into GRASS

```
#!/bin/bash

if [ $# -ne 2 ]
then
  echo "Usage: $(basename $0) directory output_maps_prefix"
  exit 1
fi

PREFIX=$2
DIR=$1

for FILE in $DIR/*.tif
do
    BFILE=$(basename $FILE .tif)
    BMAP=$PREFIX$BFILE
    r.in.gdal input=$FILE output=$BMAP memory=200

    g.region rast=$BMAP.red

    r.composite red=$BMAP.red green=$BMAP.green \
    blue=$BMAP.blue output=$BMAP.rgb
done
```

Listing B.5: Command to create one map from many others

```
g.region rast=‘g.mlist type=rast pattern="prefix_*.rgb" sep=,’
r.patch input=‘g.mlist type=rast pattern="prefix_*.rgb" sep=,’\
        output=prefix_all.rgb
```

Listing B.6: GDAL/OGR command and GRASS import command (*v.in.ogr*) for importing a shapefile

```
ogr2ogr -t_srs EPSG:32632 training_buildings \
        11_adding_training_data/Training_building_data.shp

v.in.ogr dsn=.../training_buildings/Training_building_data.shp\
         output=training_buildings
```

Table B.1: English explanations of attribute names and used Italian terms

attribute/Italian term	explanation
S/sì	yes/true
N/no	no/false
codice	id/identifier
INSCOD317/CODICE317	unique id
GSATTRBRO	gallus; broiler
GSATTRRIP	gallus; reproduction
GSATTROFP	gallus; production of FASE POLLASTRA (pre-broiler)
GSATTRODA	gallus; egg; layers in open space
GSATTRODB	gallus; egg; biological layers
GSATTRODG	gallus; egg; layers in cage
GSATTRODT	gallus; egg; layers on ground (never in cage)
GSTAVICPOT	gallus; potential capacity, number of birds
TSATTRTPRIP	turkey; reproduction
TSATTRTPCAR	turkey; meat production
TSTAVICPOT	turkey; potential capacity, number of birds
ASAVIANA	ducks
ASAVICOT	reared partridges
ASAVIFAG	pheasants
ASAVIFAR	guinea-fowl
ASAVIOCH	geese
ASAVIPER	pernici
ASAVIPIC	pigeons
ASAVIQUA	quail
ASAVIRAT	ratites (ostrich, etc.)
ASAVISTA	reared partridges
ASTAVICPOT	potential capacity, number of birds of all AS* species
AUTOCONSUMO	backyard/flock production for self consumption

Table B.2: List of farm codes and counts of animals

codice	number
035PD803	23000
040VR043	78000
055VR112	25000
059PD058	15000
065VI603	45000
085VR036	16000

```
SELECT * FROM avi_tuttelespecie_short_csv
    WHERE AUTOCONSUMO='S' AND
        (GSTAVICPOT > 1000 OR TSTAVICPOT > 1000
         OR  ASTAVICPOT > 1000)
```

The simplified result of the SQL command is in the table B.2.

Appendix C

User guide

This text is a user guide for using *i.farm.∗* modules and other related GRASS modules. The guide is specialized for RasterVet project. File names and column names usually are those used in this project. In case you want to use this guide generally, you may ignore some sections.

This guide assumes that your GRASS location is in projection UTM zone 32. However, it should work for any other projection.

GRASS vector category is used as an unique feature identifier. The text mentions column name `codice317` and other variations of this name. This column contains unique identifiers of Italian farms (identifier used in national legal or technical documents).

C.1 Map name conventions

The safest practice is to use names which begins with a letter (not a number) and does not contain any dash or dot (underscores are allowed). Other special signs are not allowed. This applies for both vector and raster maps (see C.1 for details). However, raster map names can contain dots and this practice is used by many GRASS modules. *i.farm.∗* modules internally replaces dots by underscores when necessary (e.g., column names). The reason why the naming rules are so complicated is that map names must fulfill various requirements such as be a valid file name (on various operating systems) or be SQL compliant (while non-quoted).

Table C.1: Examples of valid and invalid map names

map name	validity
`points_125650`	valid
`rv_125650.blue`	valid for a raster (not for a vector)
`125650`	valid for a raster but shall not be used
`points.125650`	invalid for a vector (valid for a raster)
`points-125650`	invalid

C.2 General process overview

The figure C.1 on the next page shows which data have to be prepared for the module *i.farm.detect.buildings*. Note that some parts were simplified.

C.3 RGB to HIS conversion

Use *i.rgb.his* for RGB to HIS conversion.

C.4 Smoothing and segmentation

You may obtain better results when you run some smoothing like *r.neighbors* (average) on HIS maps. This option is highly recommended.

Another possibility is to run segmentation. There are *i.segment* and *i.segment.xl* modules. However, you cannot use output of this modules directly and you have to do some other processing. More straight forward way is to use *r.seg* which provides smoothing based on segments. This module is available only for GRASS 6.

C.5 Texture

To generate texture map run *r.texture* with parameter `method` set to the value `corr` (produced map will be suffixed by `_Corr`). The `size` of moving window should be set to 7 and the `distance` between two samples to 1 (the default). The resolution should be set to 2 (run *g.region* with `res=2`).

C.6 Training areas

Generally, there are two maps which can be referred as training areas, namely maps with buildings and map with areas like fields, forests etc. For creating signature file (section C.7), the later one is needed.

Firstly, create training areas for you area (as a vector map). The first class (category 1) in the training areas has to be light buildings class. You can combine several existing maps together but be aware of category numbers. It is a good idea to create an attribute table and provide a column `name` containing a class name.

Secondly, convert the vector map to raster using *v.to.rast* (listing C.1 on page 94). The `use` parameter shall be set to `cat` (module converts categories to raster map values).

C.7 Signature file

At first, create an imagery group using *i.group* module. Add hue, intensity, saturation and texture map to this group (in this order).

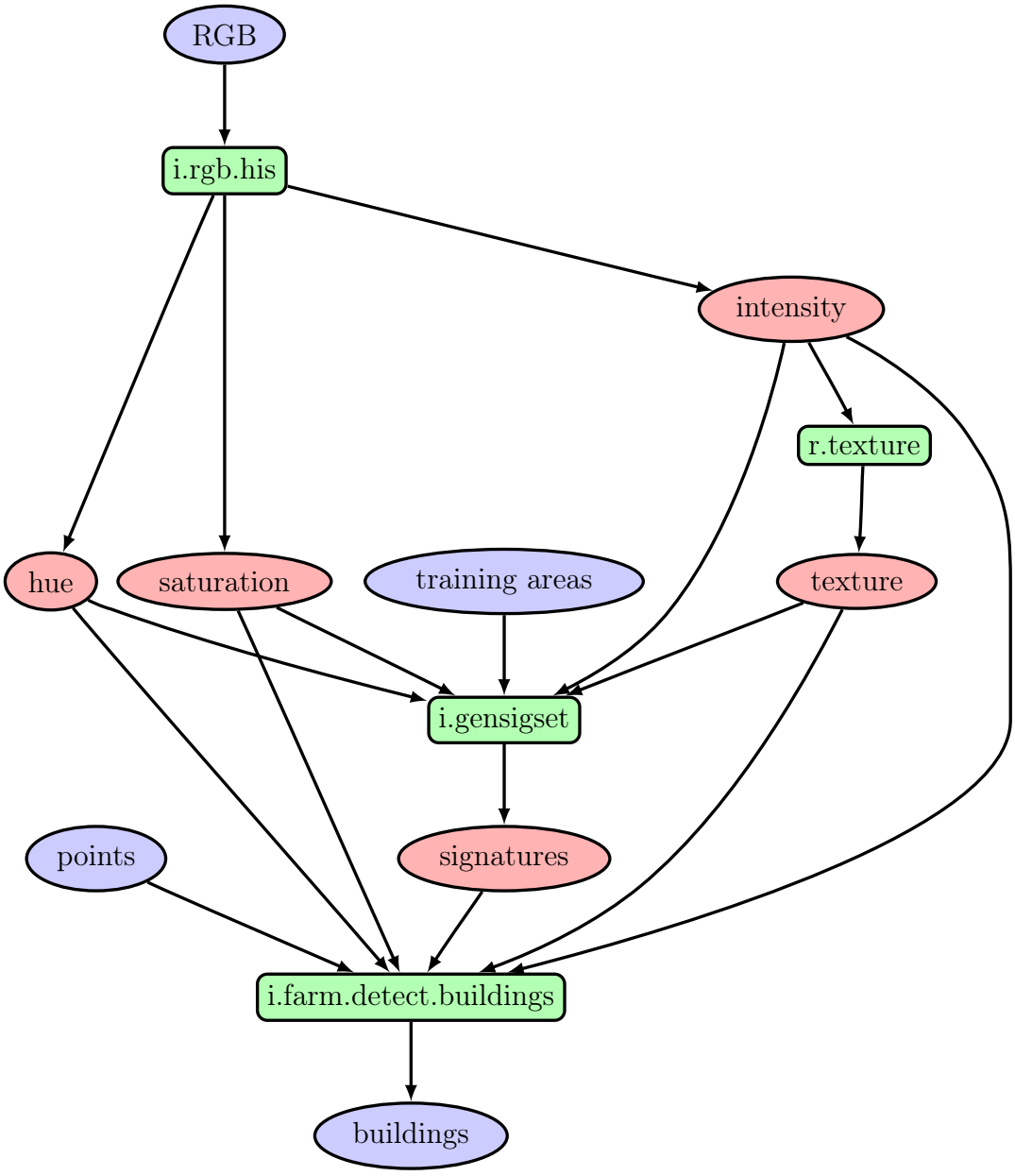


Figure C.1: General process overview (simplified). The figure shows which data have to be prepared for the module *i.farm.detect.buildings* and how (blue — input and output, red — other data, green — modules and functions; nodes are clickable)

Listing C.1: Generating signature file from training areas (vector map)

```
#!/bin/bash

# set variables
SIGFILE=farm_signatures
VECTOR_MAP=$1
RASTER_MAP=tmp_$VECTOR_MAP
GROUP=$2

# convert vector to raster
v.to.rast input=$VECTOR_MAP output=$RASTER_MAP \
          use=cat labelcolumn=name

# generate signatures
i.gensigset trainingmap=$RASTER_MAP signaturefile=$SIGFILE \
            group=$GROUP subgroup=$GROUP \
            maxsig=1

# remove temporary raster
g.remove rast=$RASTER_MAP
```

The input for *i.gensigset* is formerly created imagery group and training areas (raster map), see listing C.1. The classification module internally used by *i.farm.mask.classification* is *i.smap*.

Note that you don't have to create signature file if you already have one. If the area is consistent you can create one signature file and use it for all maps in the area.

C.8 Running the detection

A full (or relative) path to signature file have to be used. Only the name is not enough because there is no standard mechanism to copy signature files.

See listing C.2 on the next page for example of the command for detecting buildings on maps `x24100_UTM32` and `x24110_UTM32`.

Note that you need a signature file which creation is not covered by this document. However, sample signature file is provided (section C.11 on page 96).

C.9 Associating rectangles and approximate points

The association can be done through category, codice 317 or another unique id. Technically, you upload the identifier from the closest point to the each building.

Create a new column in the buildings map and use one of the commands from the listing C.3 on the next page to do the update.

Listing C.2: Example of running the module for building detection

```
i.farm.detect.buildings \
 points=approximate_points \
 output=buildings_results \
 raster=x24100_UTM32.intensity,x24110_UTM32.intensity \
 hue=x24100_UTM32.hue,x24110_UTM32.hue \
 intensity=intensity,x24110_UTM32.intensity \
 saturation=saturation,x24110_UTM32.saturation \
 texture=x24100_UTM32.texture,x24110_UTM32.texture \
 signaturefile=your/path/to/signature/file \
 window=100
```

Listing C.3: Examples commands for updating associating rectangles and points

```
# upload point's codice317 to the map buildings
v.distance to=epidemiologic_points from=buildings \
           upload=to_attr column=farm_codice317 \
           to_column=codice317

# upload point's category to the map buildings
v.distance to=epidemiologic_points from=buildings \
           upload=to_attr column=farm_cat to_column=cat

# upload point's category to map buildings (simplified syntax)
v.distance to=epidemiologic_points from=buildings \
           upload=cat column=farm_cat
```

C.10 Testing the result

For testing the result just run the *i.farm.rectangles.compare* module:

```
i.farm.rectangles.compare compared=results correct=buildings
```

From its output you are probably most interested in two numbers — detection percentage and branch factor.

Whether the approach described in C.9 on page 94 works can be tested on training (testing) data after associating by simple SQL statement similar to this:

```
SELECT * FROM training_data_buildings_with_codice
         WHERE cod_317 != farm_codice317
```

The should return zero rows. This assumes that training data already has a codice 317 column.

C.11 Sample contents of the signature file

This file may be useful if you are using similar data as used in testing when was this particular file created. However, you usually need to generate your own which is specific for your data (described the section C.7 on page 92). File contents follows.

```
title: Farms
nbands: 4
class:
 classnum: 1
 classtitle: farm_light
 classtype: 1
 subclass:
  pi: 1
  means: 127.607 177.45 15.773 0.447747
  covar:
    2695.68 648.222 219.458 -0.945445
    648.222 1140.77 119.892 -1.17374
    219.458 119.892 105.539 -0.0105882
    -0.945445 -1.17374 -0.0105882 0.0346201
 endsubclass:
endclass:
class:
 classnum: 2
 classtitle: farm_green
 classtype: 1
 subclass:
  pi: 1
  means: 115.274 91.2219 44.0977 0.302016
  covar:
```

```
    80.3819 -39.8215 6.12336 0.1857
    -39.8215 536.424 -38.4844 0.731864
    6.12336 -38.4844 101.99 0.26353
    0.1857 0.731864 0.26353 0.0290883
 endsubclass:
endclass:
class:
 classnum: 3
 classtitle: field_green
 classtype: 1
 subclass:
  pi: 1
  means: 112.05 66.319 51.9404 -0.0795546
  covar:
    18.2123 -14.9245 3.76736 -0.109167
    -14.9245 58.0749 -36.9799 0.108991
    3.76736 -36.9799 52.7251 -0.0588087
    -0.109167 0.108991 -0.0588087 0.0125169
 endsubclass:
endclass:
class:
 classnum: 4
 classtitle: field_brown
 classtype: 1
 subclass:
  pi: 1
  means: 52.6947 109.726 16.5914 0.0679418
  covar:
    179.539 -112.733 -30.2205 -0.118619
    -112.733 160.982 26.6288 0.347742
    -30.2205 26.6288 16.8158 0.0323852
    -0.118619 0.347742 0.0323852 0.0191481
 endsubclass:
endclass:
class:
 classnum: 5
 classtitle: field_brown_green
 classtype: 1
 subclass:
  pi: 1
  means: 88.2179 93.5112 18.9034 -0.0577527
  covar:
    397.995 -208.505 47.0605 -0.431645
    -208.505 458.02 -80.0147 0.85727
    47.0605 -80.0147 41.151 -0.0706995
```

```
    -0.431645 0.85727 -0.0706995 0.0229827
 endsubclass:
endclass:
class:
 classnum: 6
 classtitle: forest
 classtype: 1
 subclass:
  pi: 1
  means: 118.086 55.5145 41.1087 0.131536
  covar:
    111.783 -98.5131 32.9194 0.110017
    -98.5131 278.097 -136.085 -0.0424166
    32.9194 -136.085 130.798 0.0650385
    0.110017 -0.0424166 0.0650385 0.0172633
 endsubclass:
endclass:
class:
 classnum: 7
 classtitle: farm_red
 classtype: 1
 subclass:
  pi: 1
  means: 95.4778 88.5163 28.3079 0.275849
  covar:
    13927.3 209.616 -4.40527 0.890784
    209.616 131.412 23.891 -0.198569
    -4.40527 23.891 42.3813 -0.136383
    0.890784 -0.198569 -0.136383 0.0251074
 endsubclass:
endclass:
```

List of Figures

1.1 The basic ways a user can interact with the GRASS GIS functionality 26
1.2 The simplified building detection workflow 28
1.3 The general building detection workflow 29
1.4 Approximate points are sometimes well positioned but sometimes they are far away from farm buildings which they belong to 30
1.5 The light roof type (a) and the dark one (b) 32
1.6 The red roof type (a) and the green roof type together with light roofs (b) 32

2.1 Canny Edge Detection: (a) original image, (b) noise reduction, (c) intensity gradient image, (d) non-maximum suppression, (e) thresholding with hysteresis 36
2.2 r, θ line parametrization 37
2.3 Explanation of Hough transform 38
2.4 Basic Hough transform example showing recognizable features of a rectangle 38
2.5 Original image (a) and its transformation (b) with preserved relative scale 39
2.6 Example of original and Hough transform image. Lines in the original image are connected with the corresponding maxima in Hough image 39
2.7 Perpendicular profiles with the highlighted median profile 41
2.8 Standard deviation computed gradually from parallel profile values and indicated place where the significant value change occurs 42
2.9 Parallel and perpendicular line segments created from several edge maps are connected together in various combinations. 43
2.10 Simplified schema of rectangle detection (blue — input and output, red — other data, green — modules and functions; nodes are clickable) 44
2.11 Mask used to filter out unwanted areas where there are no buildings (areas not covered by mask are excluded — standard GRASS GIS mask behaviour) 45
2.12 Filtering rectangles which are not buildings 47
2.13 Rectangles before (a) and after (b) classification 48
2.14 Different kinds of duplicates 48
2.15 The example result of removing duplicates which were mainly based on intensity values 49

2.16 Examples for rectangles being inaccurate in length. Correct rectangles are green, detected rectangles are red. We can see small length differences. Some rectangles are longer (a) while some other shorter (b). 50

3.1 Ways to use GRASS libraries and modules; all nodes are part of the interface, so they are accessible to a programmer (note that ctypes interface and pyGRASS are in fact parts of GRASS Python library) . 52
3.2 Example output of Python profiling; the output (intermediate `dot` file) was simplified because standard graph is very large; we can see the number of calls, percentage of (absolute not processor) time spent by calling a function and percentage of time spent in function itself (self-cost) 58

4.1 Result of *i.edge* with different initial values 62
4.2 The module *i.edge* applied on intensity channel of an photograph containing various types of edges. Note how hair on the face which is hardly visible at the original photograph was detected as edges. Also note the trees and electricity pole in the background which shows what detector evaluated as edge and what not. The used parameters follow: $\sigma = 1,\ lT = 3,\ hT = 5$ 63
4.3 Line segments detected by the Hough transform 63
4.4 Maps from mlpy classifier test; (a) is a used for creating training dataset and (b) is the resulting classified dataset which were classified using values from various raster maps by *v.class.mlpy* 66
4.5 Building detection overview — part I. The first part of building detection when all raster maps are processed, rectangles are detected and vector map with attributes derived from raster maps is created (blue — input and output, red — other data, green — modules and functions) 69
4.6 Building detection overview — part II. The second part of building detection when rectangles representing buildings are extracted from vector map. In the second phase duplicates are removed. Extraction and removing of duplicates are based on attributes (blue — input and output, red — other data, green — modules and functions) 70
4.7 Detected farm buildings; the figure (a) shows successful detection in the area with other building; the figure (b) shows only partial success when roofs with color similar to the background and roof with dark spots were not detected 70

C.1 General process overview (simplified). The figure shows which data have to be prepared for the module *i.farm.detect.buildings* and how (blue — input and output, red — other data, green — modules and functions; nodes are clickable) 93

List of Tables

1.1 Absolute numbers and percentages of various colors of building roofs in the sample dataset 33
1.2 Percentages of cells and buildings areas in the sample dataset 33

3.1 The percentage of clock time spent in function and by module call for different input complexity of task (range is the size of range the module or function loops over) 56

B.1 English explanations of attribute names and used Italian terms 88
B.2 List of farm codes and counts of animals 89

C.1 Examples of valid and invalid map names 91

List of Listings

3.1 Moving variable definition out of inner loop ensures that the memory is allocated only once . . . 54
3.2 Bash script which profiles a Python script and outputs PDF and PNG files with call graph . . . 55
3.3 The fist module for the subprocess call benchmark . . . 56
3.4 The second (main) module for the subprocess call benchmark . . . 57
4.1 The testing script which shows how *v.class.mlpy* works. Is is intended to run in the GRASS North Carolina sample dataset. It generates random points and uses raster maps (landsat images) to fill their attribute tables. Then, the class is added to the training dataset. At the end, classification is performed on random points using training dataset. . . . 65
B.1 SQL create statement for temporary table with attributes . . . 85
B.2 SQL create statement to create new combined table . . . 85
B.3 Shell script to transform GeoTIFFs (directories and file names are hard coded) . . . 86
B.4 Shell script to import multiple GeoTIFFs into GRASS . . . 87
B.5 Command to create one map from many others . . . 87
B.6 GDAL/OGR command and GRASS import command (*v.in.ogr*) for importing a shapefile . . . 87
C.1 Generating signature file from training areas (vector map) . . . 94
C.2 Example of running the module for building detection . . . 95
C.3 Examples commands for updating associating rectangles and points . 95