Sheffield Hallam University

Universidad de Córdoba

Recognition of 2D-objects using RANSAC

Sonia FERNADEZ-RODRIGUEZ

Project Supervisor

Jan WEDEKIND

May 2008

Contents

List of Figures

Acknowledgements

First of all, I would like to express a massive thank you to my supervisor, Mr. Jan Wedekind, for his infinite patience and reliability. He helped me everytime I needed, providing me the knowledge and support to carry out my project work.

Moreover, I would like to express thank you to Dr. Bala Amavasai, Sheffield Hallam University, and Mr. Eduardo Gutiérrez de Ravé Agüera, Universidad de Córdoba, for giving me this chance to have this great experience.

Thanks too to my flatmates and friends, Rafa, María, Raquel, Silvia and José, for listening me and for advising me when I needed. Also, thanks for sharing with me the good and bad moments. Thanks to Bigotes for be always by my side.

Finally, thanks to my family for providing me all support and resources that I needed, specially thanks to my uncle for advising me during the writing of this report.

Chapter 1.

Introduction

1.1 Motivation

Computer vision is about building artificial systems in order to obtain information from images, interpret it and work with it later. Nowadays computer vision is used in a large number of fields; there is hardly any sector where image processing cannot be applied. There are early applications (for example, mobile robot navigation and military intelligence) as well as novel ones (for example, human computer interaction and medical image analysis). As the field of computer vision is very broad and has undergone notable progress in the last years, we will focus on the field of image processing [6]. This paper is directed to applications in the fields of micro- and nano-technology, since it is a need which is becoming more and more important in industrial tools that try to automate with precision the process of assembly for small pieces. Therefore, the feedback system could need computer vision for control it in a near future.

If an optical microscope is fitted with a digital camera, images can be analysed in real-time and the control and monitoring of tool and object becomes possible. This work is about real-time recognition of rigid objects.

This project was done within the *Microsystems and Machine Vision Laboratory* at Sheffield Hallam University. The project is in support of the PhD work on *machine vision methods for microscopes and micromanipulation* by Jan Wedekind The project makes use of HornetsEye which is an extension for the Ruby[1] programming language. The main goal of HornetsEye is to provide a more practical tool for a future use in machine vision systems.

1.2 State-of-the-art

In many cases there are several computer vision methods addressing the same problem, *i.e* , there is often a non-standard solution. Furthermore applications have to be tested in a particular hardware, so that a result forecast usually is not possible. These reasons cause some problems when we work with algorithms of image processing. Despite the fact that multiple image processing applications can be found, the state of this kind of research is still not very developed in several areas. Hence, many applications for object

1 http://www.ruby-lang.org/

recognition can be found, but in the field of microscopy limits the number of applicable solutions.

There are a significant number of studies about computer vision in general and, specifically in area of microscopy. However, there is a non-model of steps to follow in order to recognise and track an object. Despite it, these systems frequently carry out the next flow of steps (see Figure 1.1). Thus, this paper mainly focuses on the following steps: *Feature Description* and *Recognition/ Tracking* in this context of object recognition. *Feature Description* consists on determinate which are the features, that means, the pixels considered significant will be chosen. *Recognition/ Tracking* are made by means of these features and a known scene image.

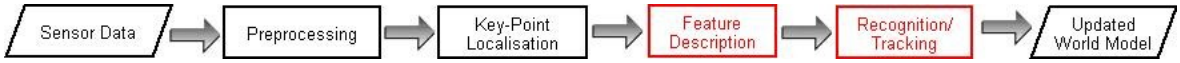

Figure 1.1: Overview of a typical machine vision system

In many situations, the interpretation of detected data becomes possible by means of predefined models. This interpretation covers two different tasks, classification and parameter estimation, which usually are dependent. The former task consists on fitting the sensed data with the model. The latter is to compute the best values for the free parameters of the model chosen.

Several techniques for parameter estimation, for instance least squares, have not internal mechanisms to detect and discard big errors, since they assume that there is a direct function that will found good data, although from the beginning, there are gross errors. However, several practical problems do not keep this assumption, called smoothing assumption [8]. On the other hand, previously to this project geometric hashing was used for the real-time object recognition of micro-objects in this team work, but the performance of geometric hashing does not scale well with increasing number of objects. To try to solve these problems, RANdom SAmple Consensus (RANSAC) is implanted in the current work [1]. RANSAC is a robust procedure used to fit figures.

Currently, RANSAC method is the option for an enormous amount of problems and has had a great impact on the structure from motion community. Moreover it is applied to fitting problems in image reconstruction. For example, interpretation of maps are being made and strategies applying to map reconstructions [6].

1.3 Context

1.3.1 MMVL

The *Microsystems and Machine Vision Laboratory*, MMVL, is a work group of the Materials and Engineering Research Institute at Sheffield Hallam University, UK [1].

The leading works are associated with the design, development and implementation of machine vision techniques focused on several real-time and non real-time applications, such as, micro-robotic systems, micro-manipulation, microscope imaging [2].

1.3.2 HornetsEye

HornetsEye is a Ruby-extension for developing video processing and real-time computer vision software using GNU/Linux platform. This new class incorporates into Ruby a better way of working with computer vision, since it makes easy the processing of image- and video-I/O with RMagick, Xine, firewire digital camera, and video for Linux. HornetsEye provides a collection of algorithm, such as, edge detection and corner detection, which allow a simple search of the main pixels of the images to this program [1].

Moreover, this extension provides several features, for instance, “Ruby element-wise array operations” where these operations are provided by MultiArray and “XVideo widget” which is a Qt4-QtRuby widget for displaying videos using XVideo hardware acceleration. Several features are under testing while others are considered to work stable. However the features do not work equally well on different platforms [4].

HornetsEye is a free software distributed under GPLv3 (General Public License[1]).

1 http://www.gnu.org/licenses/

Chapter 2.

Project Outline

This chapter presents the purpose of this project, how this aim was achieved and the equipment used during the project's development.

2.1 Project definition

The purpose of my work was to develop a software which is able to recognise rigid objects in 2D for real-time object recognition. This software must employ the RANSAC algorithm. Moreover, this software must be implemented with the Ruby programming language.

First of all a literature survey about computer vision in general and RANSAC in particular was performed. Notable papers and books about RANSAC are [6], [7], [8] and [9] where I could find different kinds of RANSAC implementation.

2.2 Methodology

The project objectives are given in *chapter 3.1*. A detailed description of the RANSAC algorithm is given in *chapter 3.2*. *Chapter 3.3* shows how the RANSAC algorithm can be applied to the problem of 2-D rigid object recognition and test results are given.

The RANSAC algorithms for object recognition consists of the following steps which are going to be explained in detail later:

- Feature extaction.
- Feature descriptor.
- Correlation coefficient.
- RANSAC (for either translation only or translation and rotation)

Finally, it will be made some examples of execution and conclusions will be expounded.

2.3 Equipments

2.3.1 Hardware Specifications

To develop this project a laptop with the following features was used:

- Intel Core Duo T2300 Processor, 1660 MHz
- 1GB DDR2 SDRAM Memory
- 100 GB Hard Disk Drive
- Graphic card: NVIDIA GFORCE GO 7300 MB

2.3.2 Software Specifications

The software of this project has been implemented and tested under the GNU/Linux Kubuntu distribution version 7.0 using the **Ruby** programming language. Ruby is an interpreted and imperative language. This language is object oriented with single dispatch and single inheritance. Other features of Ruby are dynamic typing, support for modules, which can be imported into a class as mixins using the "include" statement (*e.g.* "include HornetsEye"). Other features of Ruby are reflection which lets you query the existence of methods and classes at runtime and generate method calls [10]. This makes Ruby a multi-paradigm language.

Chapter 3.

Design

3.1 Work objectives

The main goal of this project is to use the RANSAC algorithm in order to recognise rigid 2D objects in real-time. RANSAC must be applied to at least three degrees of freedom problems.The project goal was achieved as follows:

- Implement an algorithm for selecting the main points of both model image and scene image, *i.e.*, **feature extraction**.

- Implement a **feature descriptor** and an associated similarity measure.

- Implement the **RANSAC algorithm** in order to search the best transform matching scene pixels with model pixels. A random sample of two model features and their best corresponding features in the scene is selected and checked for geometric consistency. An affine transform is determined which maps the randomly selected features on their corresponding features in the scene. Then, a consensus with the most important pixels will be made to confirm a suitable fitting with the obtained pattern.

- Finally, the algorithm is tested on simulated and real data.

3.2 RANSAC

3.2.1 Introduction

A large number of image processing and computer vision activities are performed by fitting a set of data to a suitable model. Moreover, registration of two partly overlapping images taken from different observations is a significant task in 3D computer vision [3]. Thus, some of the advantages which can be found in Random Sample Consensus are the following: it is a paradigm using to fit a model to experimetal data, it allows interpretating and smoothing data which contain a great number of gross errors and it does not require initial estimates of motion parameters and can solve the partly overlapping 3D problem.

To understand the RANSAC algorithm in a better way, an overview of the method is shown in the next diagram:

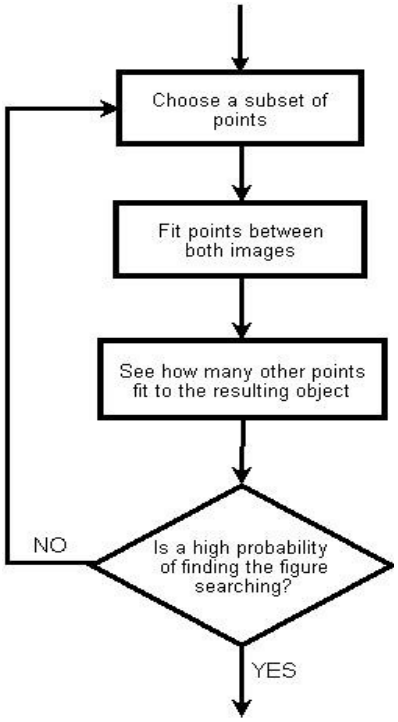

Figure 3.1: Overview of RANSAC algorithm [6]

Where "fitting points" means to establish a relationship between the subset of obtained points and a subset of the searching image pixels.

Therefore, the main aim of RANSAC is to search a subset of good points by means of a random sample of points, where a good point, also called "inlier", means that this point can be considered to fit the sensed data to a predefined model image.

3.2.2 RANSAC behaviour in simple case (fitting line)

To appreciate how the algorithm works, it is necessary to definy several concepts, such as *sample*. A *sample* consists of sets of points drawn in a random and uniform manner

from the data set. Each sample should contain a minimum number of points necessary to fit with the model image, for instance the need of two points if a line is tried fitting. Let suppose that *n* data points need to be drawn and the fraction of these points which are good is *p*. The value expecting of number of draws necessary to get one point will be the following [6]:

$$E[k] = 1\mathrm{P}(\textit{good sample doing one draw}) + 2\mathrm{P}(\textit{good sample doing two draws}) + \ldots = p^n + 2(1-p^n)p^n + 3(1-p^n)^2 p^n = p^{-n} \quad (3.1)$$

However, it is very usually to deal with data where *p* is unknown. Hence, each matching attempt has information about *p*. If *n* sensed points are necessary, then it can be assumed that the probability of a good matching is p^n. To estimate *p*, several observations have to be taken into consideration the fitting attempts. Therefore, it is suggested beginning with a low estimation of *p*, then produce a succession of tested matchs, and finally improve this value of *p*. When there are more fitting attempts than they are needed, the process can finish [6].

For example, if it is being looking for fitting a line in a set of points, it is possible to start with a probability of 50%, then it is checked if a good sample is found to fit it to a line with a fifty per cent of the data points. Later, it could be taken a 70% of probability, and so on, until a high probability is found.

Another problem of this algorithm is to determinate when a point can be considered as “good”. For this reason, it is needed to establish whether a point lies near to a line fitted to a sample or not. The distance between the point and the fitted line is determined. Later, a test is made between that distance and a threshold *t*, which we defined so that if the distance is below the threshold, then the point will be considered as an inlier (good point). Defining this parameter usually is a part of the modelling process. Overall, getting a value for this parameter is relatively simple. The paremeter is frequently established by attempting a few values and seeing what happens, although there are other approaches to estimate it [6].

Currently, there are several algorithms based on RANSAC and a huge literature about it, since fitting is a problem that happens in many contexts. This section only tries to make you understand the behaviour of RANSAC and in the next section will be explain the algorithms of RANSAC as applied to this project and the required steps for improving the performace of the algorithm for object recognition in real-time.

Example of RANSAC algorithm [9]

To better understand it is shown the next example in a simple case of fitting a line. Let the following points with different intesities, shown in the Figure 3.2.

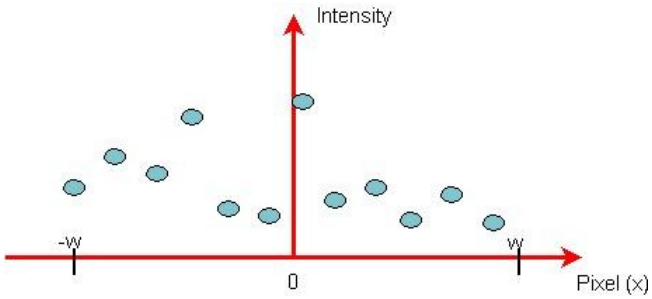

Figure 3.2: Example RANSAC. Initial pixels

1. Objectives

The goals of this sequence of steps in RANSAC are to find the inliers in a section of ratio *w* and fit a pattern, in this case a polynomial, to only the inliers pixels.

2. Variables

Previously, it will be defined the required variables. A pixel is determinated by its intesity. *Inliers* are the points whose distance to the pattern is smaller than a given threshold *t*. *Outliers* are those points whose distance is bigger than the threshold (see Figure 3.3). In the figure below pixel *g* and *d* would be outliers and the others inliers. The probability of success of fitting a large amount of pixels is called *tp* and the fraction of these points which are inliers is *p*. The degree of the polynomial will be *n-1*, where *n* is the number of pixels to choose, particularly for a line $n = 2$ since it is needed to find two coefficients in the equation of a line ($ax + by = k$).

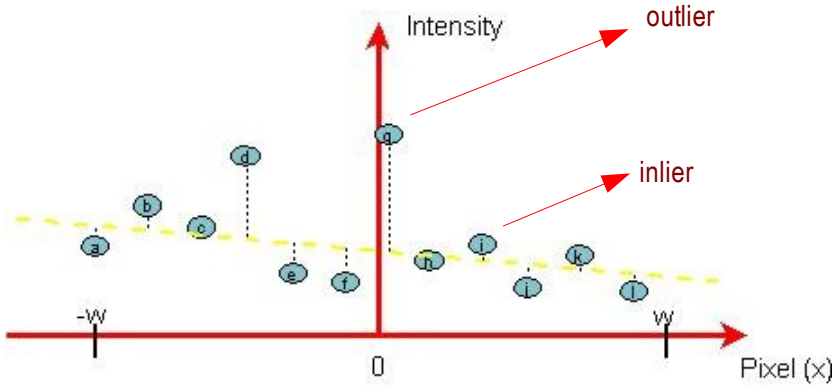

Figure 3.3: Example RANSAC. Outliers & inliers

3. Method

1. Randomly select $n = 2$ pixels from the section $[-w, w]$.

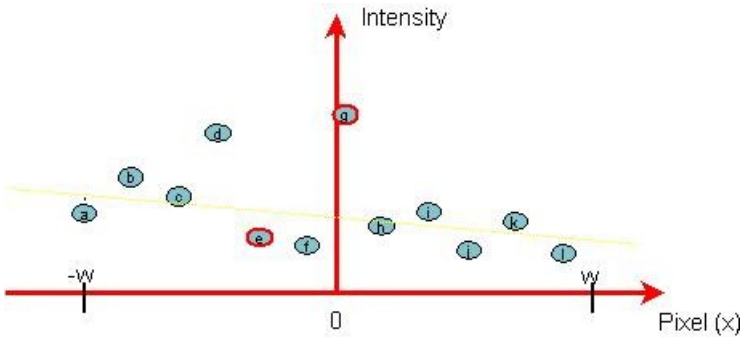

Figure 3.4: Example RANSAC. Chosen pixels (e, g)

2. Fit the polynomial conecting both pixels chosen.

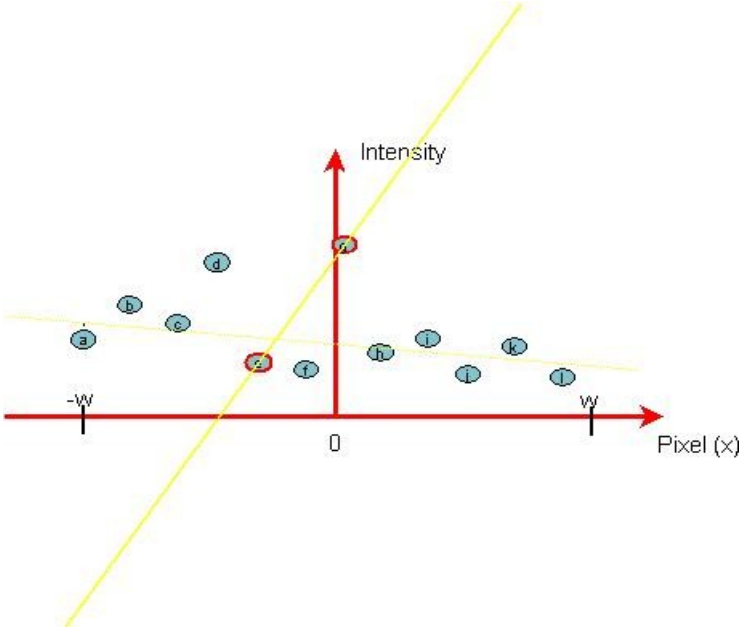

Figure 3.5: Example RANSAC. Fit line (e-g)

3. Count pixels with vertical distance less than threshold, *t*.

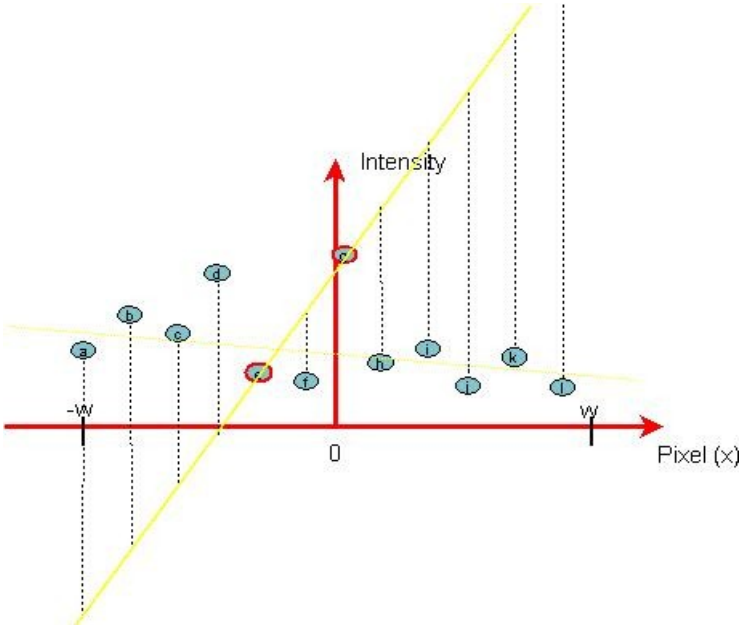

Figure 3.6: Example RANSAC. Distances to fitted line (e-g)

In this situation, only *e*, *f*, *g* would be good points.

4. If there are not "enough" inliers, **repeat** all steps but do not do it more than *T* times.

$$p = \frac{inliers}{pixels} = \frac{3}{12} = 0.25 \qquad (3.2)$$

Thus, 25% of inliers can be said that it is not enough inliers. Therefore, it would be necessary to repeat the algorithm.

1. Randomly select *n* pixels from the section [-*w*, *w*].

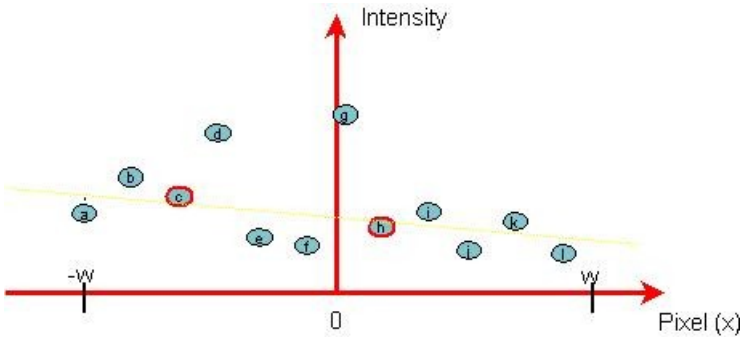

Figure 3.7: Example RANSAC. Chosen pixels (c, h)

2. Fit the polynomial conecting both pixels chosen.

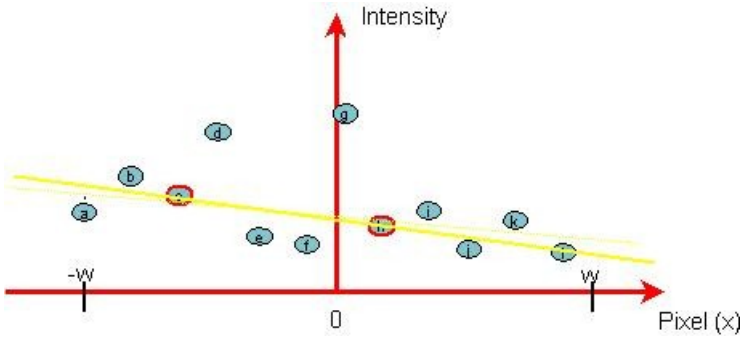

Figure 3.8: Example RANSAC. Fit line (c-h)

3. Count pixels with vertical distance less than threshold, *t*.

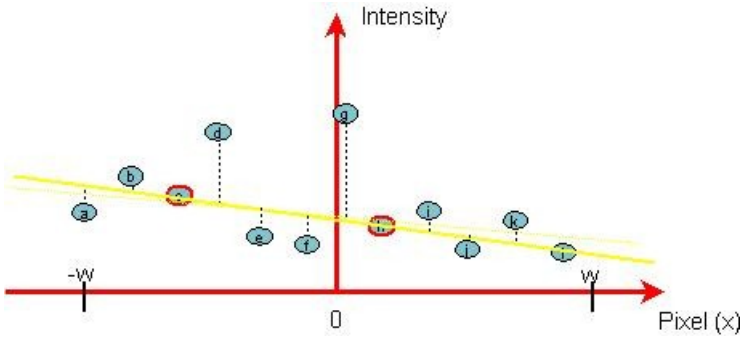

Figure 3.9: Example RANSAC. Distances to fitted line (c-h)

As one can see in the Figure 3.9, only two pixels could consider as bad point, *d* and *g*. Here, there are 10 good points.

4. If there are "enough" pixels, **stop** and label these good pixels as inliers and calculate the best line to fit all inliers.

$$p = \frac{inliers}{pixels} = \frac{10}{12} = 0.83 \qquad (3.3)$$

83% of inliers could be an enough amount of inliers. Therefore, the algorithm could finish here and label these good points. Then, calculate the best line to fit these inliers.

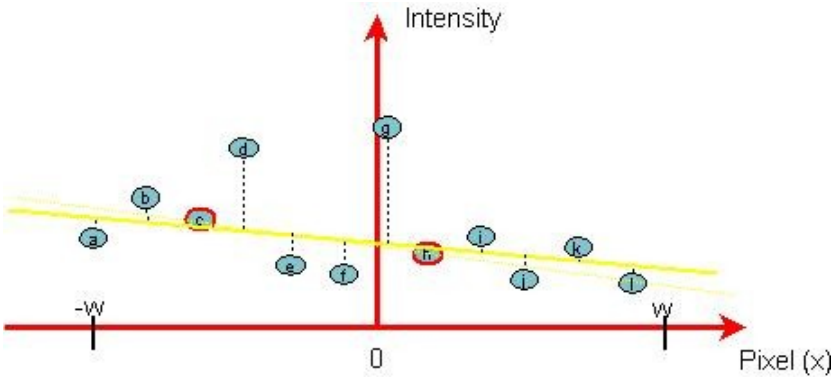

Figure 3.10: Example RANSAC. Final line to fit points

Therefore, the followed steps in this example of RANSAC are:

Repeat at most *T* times:

1. Randomly select n pixels
2. Fit (*n*-1)-degre polynomial
3. Count pixels whose vertical distance from polynomial is < *t*
4. **If** *p* is considered as good amount, **exit loop** and
   - Good points are labeled as inliers
   - Fit polynomial to all inlier pixels

where:

n = *necessary pixels*
p = *fraction of inliers*
t = *fit threshold*
tp = *success probability*

Thus, this algorithm may be followed to find structures for polynomials.

As it can be observed, there are several complexities in practical situations, such as determining distances, ensuring that there will be not too many outliers over inliers and deciding what to fit in the first place.

3.3 Method for object recognition

As one can see above, in the *section 3.1,* for object recognition this paper will follow different steps which are explained in this section. First of all, suppose that there is an object and several images of this object in different positions in a video, so that for each step two images will be considerred, called the *model image* and *scene image*. The former is an image whose features to be fitted (original object). The latter is an image of the object which has undergone translation and/or rotation in the plane.

At the beginning several feature locations are selected to reduce the amount of data. Matching the model and scene image directly would be inefficient and execution time would be too large for fulfilling real-time constraints. In the following subsection, it will be explained how a subset of data points are selected as point features. Then, the feature descriptor and the comparison between model features and scene features are presented. In the last section of this part, the two RANSAC algorithms used in this software are explained.

3.3.1 Features extraction

In digital image processing, the less point features are choosen the better. Moreover, to track not all part of a frame contains complete motion information due to this, several solutions are proposed by researchers, such as tracking corners or windows with a high spatial frequency content [11]. Therefore, the corner-points will be selected as main points in this case and these main points will be called "features". Hence, these features make easier and faster the motion detection and object recognition. Currently, diverse kinds of corner detection algorithms can be built. In HornetsEye [4], there are implementations of three corner-detection algorithms: The Yang et al., the Harris-Stephens, and the Kanade-Lucas-Tomasi corner-detector.

In this report, the **Kanade-Lucas-Tomasi algorithm** (KLT)[1] is used. This algorithm locates good features by means of analysing the minimum eigenvalue of each 2 by 2 gradient matrix. The features are tracked with a Newton-Raphson technique of minimizing the difference between the two windows [5].

An example is shown below, where one can see how the original image, Figure 3.12, is reduced to an image, Figure 3.11, where the number of significant pixels is smaller. Note that these significant pixels allow to recognise this image.

Figure 3.12: Initial image. Model image

Figure 3.11: Image using KLT over Model image

1 KLT is a method for searching good features in the computer vision. The source code is in the public domain, available for both commercial and non-commercial use.

After computing the feature image with the KLT algorithm, feature locations to track the object motion are selected by **Non-Maxima-Suppression** (NMS). Later, a better efficiency and less execution time will be obtained with NMS.

In this implementation of the NMS algorithm a mask is used in order to select the neighbouring pixels which are located around a potential feature location:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The selected points will be the previous KLT-points which fulfill the following conditions. First, the point chosen must have the maximum value in the pixels of its outline, that is, in the mask. Second, that point has to be equal or bigger than a threshold in order to limit the search and reduce execution time.

Repeat for all image pixels:

1. **If** *pixel[x,y]* >= *threshold* **and**

   pixel[x,y] > maximum point of its mask:

   - *Pixel[x,y]* is included as a feature of the image

where:

pixel[x,y] = intensity of colour in the position[x,y] of the image

threshold could be half *maxPoint.*

Regarding the previous example, if to the Figure 3.11 is applied the NMS algorithm, it is obtained the Figure 3.13. As one can see, these features let recognise the object as well.

Figure 3.13: Feature pixels

3.3.2 Feature descriptor

Once, the features of the image have been selected, it is necessary to relate pixels of different images. In this work for describing these features will be used a histogram for each feature which shows the amount of each colour.

The pixel's colour is usually represented by a number between 0 and 255, where the former is black and the latter is white. A histogram of 256 possible colours would be very large and highly sensitive to noise. Therefore, the image is normalised to a colour range between 0 and 15, *i.e.*, 16 grey values.

A feature descriptor is introduced made by means of a histogram of the grey values in the vicinity of each feature. Here the histogram has 16 bins. A feature is taken and a specific number of pixels around it. For example, if we take 3 pixels around the feature pixels we will obtain de following shape:

X	X	X	X	X	X	X
X	X	X	X	X	X	X
X	X	X	X	X	X	X
X	X	X	F	X	X	X
X	X	X	X	X	X	X
X	X	X	X	X	X	X
X	X	X	X	X	X	X

Figure 3.14: Template for histograms

where *X* is a colour represented by a number between 0 and 15 and *F* is a the feature represented by a number between 0 and 15 as well.

Then, the pixels of each luminosity are counted (*e.g.* 8 pixels with colour 0 or black, 1 pixel with colour 1, and so on until the colour 15).

```
[24, 20]=>[26, 0, 1, 0, 0, 1, 2, 1, 1, 0, 1, 0, 0, 0, 0, 16]
```

Figure 3.15: Example of histogram

3.3.3 Correlation coefficient

For recognising an object, a pattern is required in order to find the relationship between point pairs of both model and scene images. The relationship will depend on the scene motion. The correlation coefficient will be used in order to look for any linear relationship between the model features and the scene ones.

This coefficient is a real number which measures the degree to which both histograms are related. The result will be any value between -1 and +1, in which the sign means a direct relationship when the sign is positive, or an inverse relationship when the sign is negative. Moreover, if the value is close to 1, it means a perfect linear relationship, but if the value is nearby to 0 means non linear relationship.

The correlation coefficient, r_{xy}, is obtained by means of the covariance of x and y, S_{xy}, the variance of x, S_x, and the variance of y, S_y, using the following formula:

$$r_{xy} = \frac{S_{xy}}{S_x S_y} \tag{3.4}$$

where:

$$S_{xy} = \sqrt{\sum XY - \frac{\sum X \sum Y}{n}} \tag{3.5}$$

$$S_x = \sum X^2 - \frac{\left(\sum X\right)^2}{n} \tag{3.6}$$

$$S_y = \sum Y^2 - \frac{\left(\sum Y\right)^2}{n} \tag{3.7}$$

Simplifying the expression, the formula of the coefficient will be:

$$r_{xy} = \frac{n\sum XY - \sum X \sum Y}{\sqrt{\left(n\sum X^2 - \left(\sum X\right)^2\right)\left(n\sum Y^2 - \left(\sum Y\right)^2\right)}} \quad (3.8)$$

In this paper, the correlation coefficient is applied to the histogram of each scene feature with the histogram of each model feature. After the coefficient is found, the more significant coefficients which may be considered as the bigger ones than 0.5 are stored in list in ascending order.

The most significant coefficient, that is the biggest one, for each scene feature with the respective model feature will be used subsequently. However, this algorithm is not sufficient for object recognition, since the most significant coefficient may not be the matched pixel with the model, for example, two features of scene might have similar histograms. Thus, this algorithm is only used for improving the performance of the next one, RANSAC algorithm.

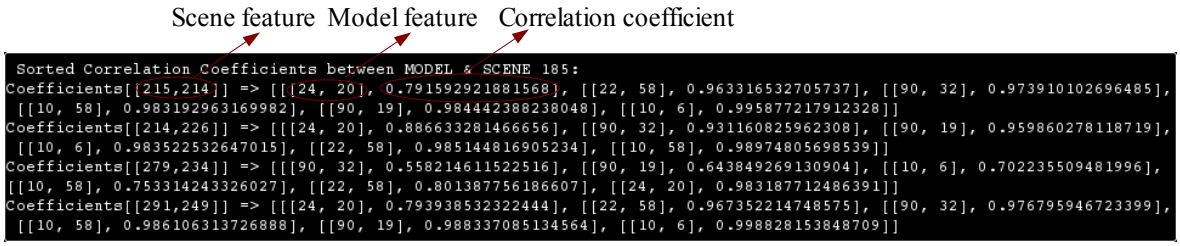

Figure 3.16: Example of correlation coefficients between a Model and a Scene image

3.3.4 RANSAC for translations

As one can see above, RANSAC has different applications in many fields. This project is focused on recognition of rigid objects which are moving with three degrees of freedom: translation in x-direction, translation in y-direction, and rotation. Therefore the distances between features are invariant, *i.e.* we are restricting ourselves to isometric transformations. In this section, translations are analyzed, and in the next section, rotations will be considered.

When we only have to take translations into account, it is possible to find the points of a figure which moved. Two pieces of information are necessary in order to find the points of the moved object. First, the **coordinates** of the initial image features, *model* features, are needed. Secondly, it is necessary a **vector** which describes the displacement for each of these features. As the translation is a displacement along straight line, a single

vector will describe the motion of all points. This vector will be called *displacement vector* and this vector is given by two coordinates, *[dx, dy]*, since we are working in 2D space; these vector coordinates specify the direction of motion and its magnitude [13].

Affine transformations can be represented with matrices, for that homegeneous coordinates are used. This means denoting a vector [x, y] as [x, y, 1]. Using this technique, a translation in 2D can be given by [14]:

$A \ \ translation \ \in \ \mathbb{R}^2 : [x, y] \ \rightarrow \ [x+dx, y+dy]$ can be represented as:

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+dx \\ y+dy \\ 1 \end{bmatrix} \qquad (3.9)$$

that is:

$$T\,P1 = P2 \qquad (3.10)$$

where column vectors are the homogeneous coordinates of the two points, *P1* and *P2*, and the matrix is translation one, *T*.

For example:

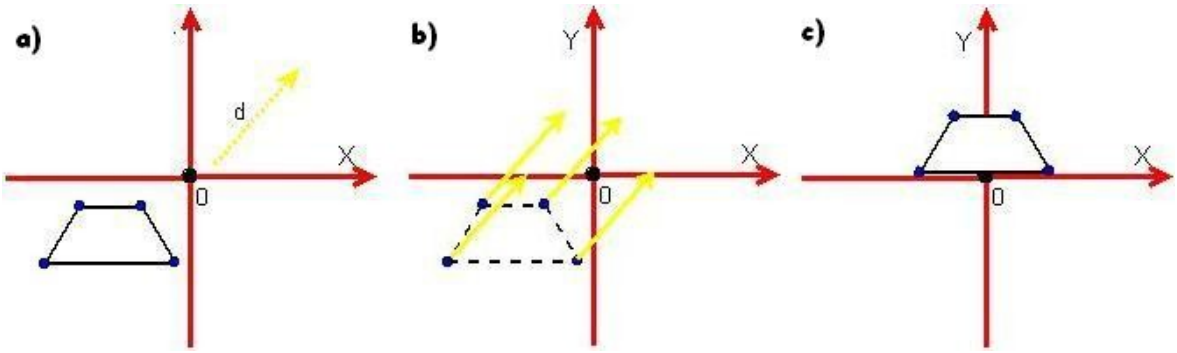

Figure 3.17: Example of translation a figure

As one can see in the Figure 3.17 all points of the figure are translated in the same direction and with the same magnitude.

In function of the area where we are working, it is used a different pattern to develop RANSAC. Therefore, *displacement vector* is used here in order to match the features of the *model* with the *scene* features, the latter are those which were shifted.

SHIFT-RANSAC ALGORITHM

Given:

distance = between Point1(modelFeature[x, y]+[dx ,dy]) and Point2(sceneFeature)

pi = probabiliy of finding inliers

T = maximum number of times to execute the algorithm

threshold = possible error in the distance between two points

tp = minimum probability of finding inliers

Repeat at most *T* times

1. Randomly select a feature of scene image
2. Calculate *displacement vector [dx, dy]* using correlation coefficient
3. **Repeat** for all model features (consensus):
   - Calculate the *distance* between the scene features and the points obtaining by means of features model and *[dx, dy]*
   - **If** *distance* <= *threshold*, number of inlier is incremented
4. **If** pi >= tp, **exit loop and** *[dx, dy]* is fit as the good motion vector

3.3.5 RANSAC for rotations

Rotation also allows to find the points of a figure which moved, since rotation is an affine transformation as well (an isometry to be more exact). Therefore, the same as translation, the rotation motion can be represented with matrices using homogeneous coordinates. A piece of information is necessary to find the points of the moved object. As a rotation is a change in the orientation of its axis, that piece is the **rotation angle**, θ. It is considered the rotation angle as those followed from the initial position, *i.e.*, the origin of the axis, to final position. The matrix position respect to the origin can be represented as [14]:

$$\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} xcos\theta - ysin\theta \\ xsin\theta + ycos\theta \\ 1 \end{bmatrix} \quad (3.11)$$

that is:

$$R(\theta)P1 = P2 \quad (3.12)$$

where column vectors are the homogeneous coordinates of the two points, *P1* and *P2*, θ is the rotation angle and the matrix is rotation one, *R(θ)*.

To obtain any point of the final position is only required to apply the formula (3.12). In contrast, if it is wanted to find a point of the first position by means of the final position, the inverse matrix will be applied (laws of matrices).

On the other hand, several affine transformations can be composed, for example if a image suffers two rotations the result of the transformation will be calculated using:

$$R(\theta_1)R(\theta_2) = R(\theta_1 + \theta_2) \quad (3.13)$$

If the rotation is not around the origin, it is in other point, it is necessary the next steps:

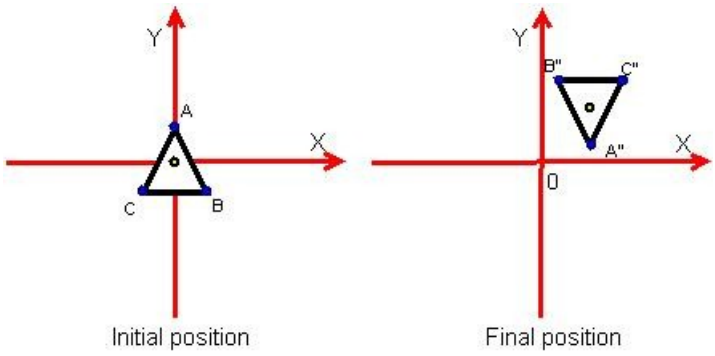

Figure 3.18: Example of rotation not around origin

1. Rotation of the figure in the origin.

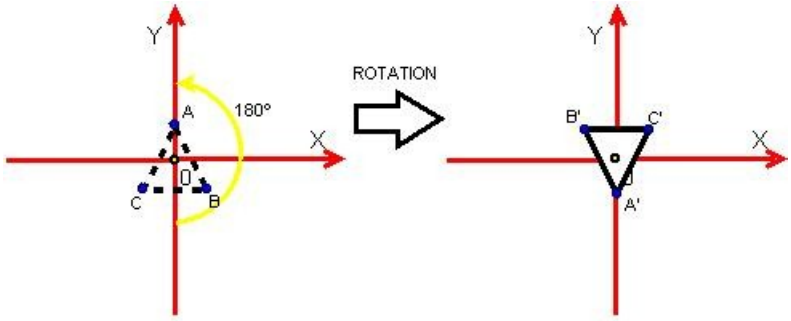

Figure 3.19: Example. Rotation of the figure around origin

$$A' = R(\pi)A$$
$$B' = R(\pi)B \qquad \text{where} \qquad R(\pi) = \begin{bmatrix} \cos(\pi) & -\sin(\pi) & 0 \\ \sin(\pi) & \cos(\pi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$C' = R(\pi)C$$

2. Translate the figure $[t_1, t_2]$.

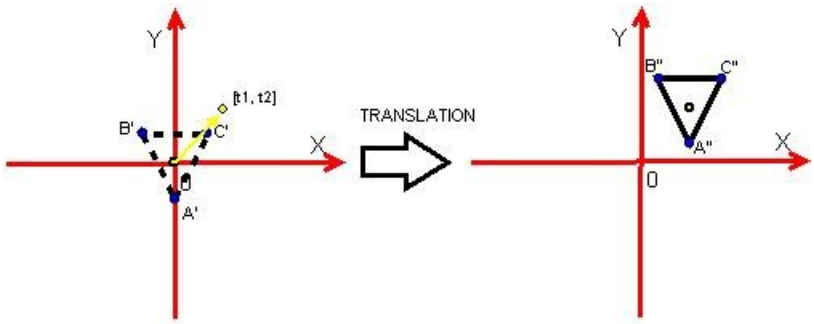

Figure 3.20: Example. Translation of the figure

$$\begin{aligned} A'' &= TA' = T(R(\pi)A) = TR(\pi)A \\ B'' &= TB' = T(R(\pi)B) = TR(\pi)B \\ C'' &= TC' = T(R(\pi)C) = TR(\pi)C \end{aligned} \qquad \text{where} \qquad T = \begin{bmatrix} 1 & 0 & t1 \\ 0 & 1 & t2 \\ 0 & 0 & 1 \end{bmatrix}$$

and it is simplified to a matrix by composing the two matrices TR,

$$TR(\theta) = \begin{bmatrix} cos\theta & -sin\theta & t_1 \\ sin\theta & cos\theta & t_2 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

This matrix, *TR*, is applied to find the pattern searching in RANSAC but it is similarly directed to vectors. The next example illustrates performance of the motion with matrices.

Extended example:

Suppose that there are two images which are determined by a vector that joins two points (see Figure 3.21), the composition of all operations is made with the different matrices. For example, to find a feature of the scene matching to a model feature, where it is given:

- a **model feature**, $B = [x_b, y_b]$,
- a vector CA from the model, $[dx_{CA}, dy_{CA}]$, which is calculated by means of two model features, C and A,
- and a vector C"A" from the scene, $[dx_{C''A''}, dy_{C''A''}]$, which is calculated by means of two scene features, C'' and A''.

These two vectors will allow to obtain the **scene rotation matrix** and the **rotation matrix of the model**.

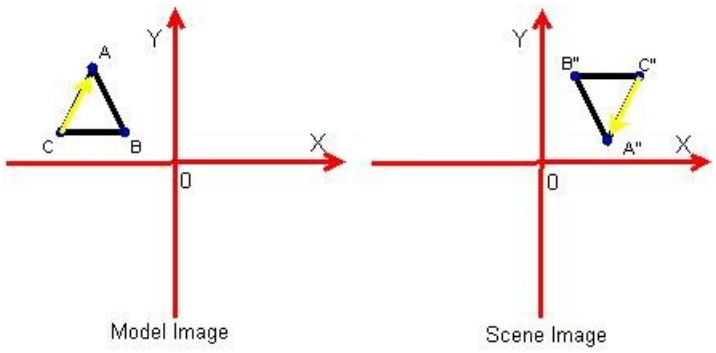

Figure 3.21: Example of rotation and translation a figure

The steps that it should be followed in this example are:

1. Calculate the rotation matrices of both scene and model.

Pattern for model image:

$$RT_m(\theta_m) = \begin{bmatrix} \cos(\theta_m) & -\sin(\theta_m) & t_{m1} \\ \sin(\theta_m) & \cos(\theta_m) & t_{m2} \\ 0 & 0 & 1 \end{bmatrix} \qquad (3.15)$$

where:

$$\begin{bmatrix} t_{m1} \\ t_{m2} \end{bmatrix} = \begin{bmatrix} \frac{x_C + x_A}{2} \\ \frac{y_C + y_A}{2} \end{bmatrix} \qquad \theta_m = \arctan\left(\frac{dy_{CA}}{dx_{CA}}\right) \qquad \begin{aligned} dy_{CA} &= y_A - y_C \\ dx_{CA} &= x_A - x_C \end{aligned}$$

Pattern for scene image:

$$RT_s(\theta_s) = \begin{bmatrix} \cos(\theta_s) & -\sin(\theta_s) & t_{s1} \\ \sin(\theta_s) & \cos(\theta_s) & t_{s2} \\ 0 & 0 & 1 \end{bmatrix} \qquad (3.16)$$

where:

$$\begin{bmatrix} t_{s1} \\ t_{s2} \end{bmatrix} = \begin{bmatrix} \frac{x_{C''} + x_{A''}}{2} \\ \frac{y_{C''} + y_{A''}}{2} \end{bmatrix} \qquad \theta_s = \arctan\left(\frac{dy_{C''A''}}{dx_{C''A''}}\right) \qquad \begin{aligned} dy_{C''A''} &= y_{A''} - y_{C''} \\ dx_{C''A''} &= x_{A''} - x_{C''} \end{aligned}$$

2. Translate the pixel of the model image to the origin, *i.e.*, in opposite direction and rotate in opposite way, which is made with the inverse rotation matrix of the model.

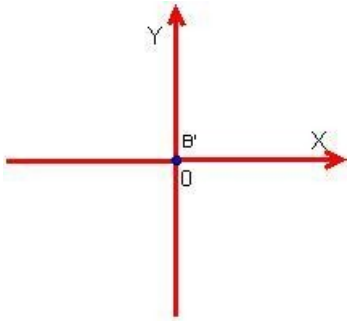

Figure 3.22: Example. Translation of model pixel, B

$$B' = RT_m(\theta_m)^{-1} B$$

These operations translate the point B to the origin.

3. Translate and rotate the resulted pixel of the image using the matrix of the scene.

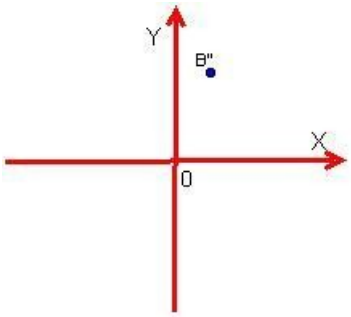

Figure 3.23: Example. Rotate the figure and obtain the final point

$$B'' = RT_s(\theta_s) B' = RT_s(\theta_s) RT_m(\theta_m)^{-1} B$$

All multiplications of matrices allow to find the point of the scene, B''.

To apply RANSAC to rotation and translation, the next algorithm is used here. The pattern for finding points from a figure to moved another is the rotation matrices seen above.

ROTATION-RANSAC ALGORITHM

Given:

pi = probabiliy of finding inliers

tp = minimum probability of finding inliers

T = maximum number of times to execute the algorithm

threshold = possible error in the distance between two points

distance = between Point1(calculated with R_s and R_m) and Point2(sceneFeature)

dm = distance between the two model features

ds = distance between the two scene features

Repeat at most *T* times

1. Randomly select two different features of scene image
2. Select the most suitable matched model features which is taken from the correlation coefficient
3. Calculate the distance between the two points of each image, *dm* and *ds*
4. **If** |*dm-ds*| >= *thresholdDist* **return** to step 1
5. Calculate both β and φ rotation angles and the vector of each pair of points, *tm* and *ts*
6. Calculate a pattern with the scene rotation matrix, *Rs,* and model one, *Rm*
7. **Repeat** for all model features (consensus):
   - Calculate the scene point using rotation matrices, *Rs* and *Rm*
   - Calculate the distance between the scene features and the points obtaining by means of matrices
   - **If** *distance* <= *threshold*, number of inlier is incremented
8. **If** pi >= tp, **exit loop and**
   Rs and *Rm* are fitted as the good rotation matrices

Chapter 4.

Results

4.1 EXECUTION EXAMPLES IN SIMULATED DATA

First of all, the software was tested in simulated data and subsequently, in real data. Therefore, this section shows some tests made in translation motion and later, rotation case will be illustrated.

Hence in the next examples the image that is wanted to recognise is the one shows in Figure 4.1.

Figure 4.1: Model Image

4.1.1 Example 1: Shift-RANSAC

In this part, some results of object recognition with two degrees of freedom are shown. Tests were made over a video where the object ungerwent only translation. The software was executed three times for each case. Thus, the next table shows the number of detected frames by the program, the amount of frames where the object was recognised and finally, the rate of object recognition each time.

Using an algorithm where the maximum number of times is 200 (T = 200) and a threshold of probability of finding inliers is 90%, the next results were obtained:

Times	Total Frames	Good Results	% Good Results
1	125	125	100%
2	125	125	100%
3	125	125	100%

As you can observe from the table, results obtained were very good, since every time a translation was made in simulated data, the object was well recognised.

Then, the example illustrates how the displacement of the model image is estimated in a series of frames (in this case it is shown from frame 100 to frame 107). You can show, the model figure is recognised in each frame, although the position is different to the initial one. To observe this object recognition, a red frame is drawn by means of the results of displacement vector and the model image.

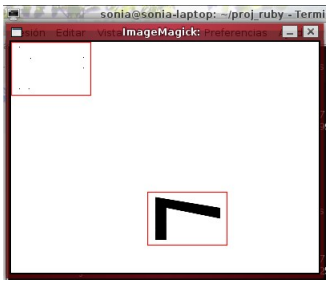

Figure 4.3: Example of translation. Frame 100

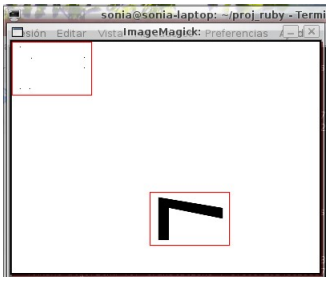

Figure 4.4: Example of translation. Frame 101

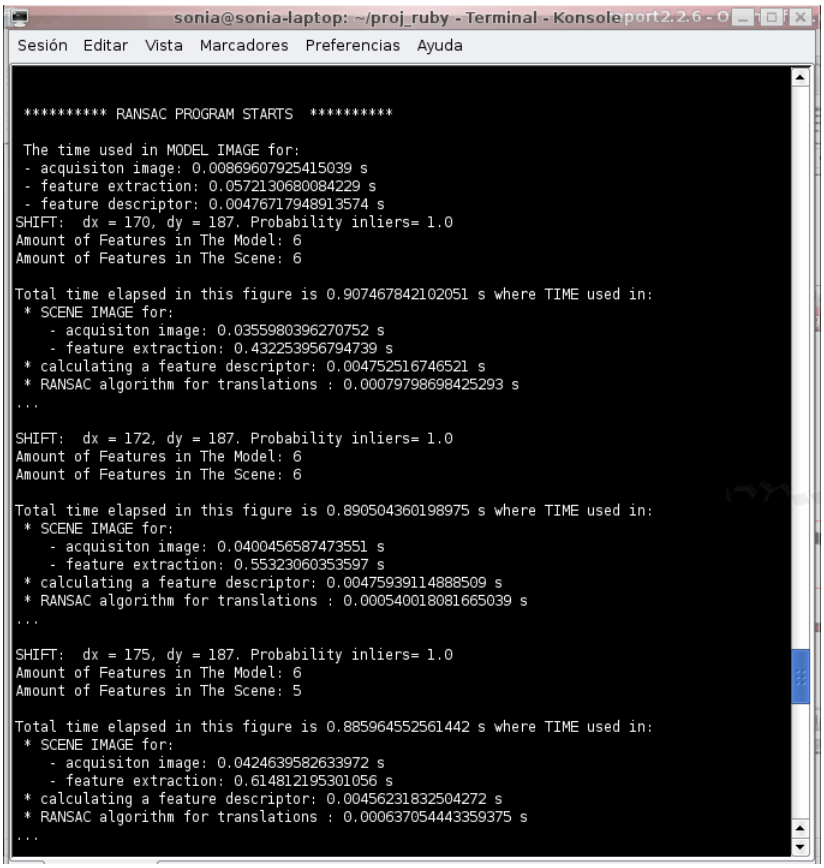

Figure 4.2: Example of translation. Program features in Model and Scene Frames [100, 102]

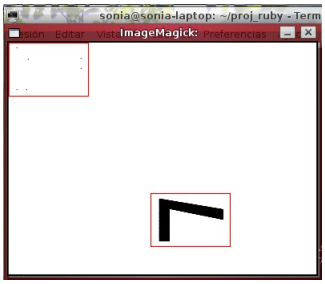

Figure 4.6: Example of translation. Frame 103

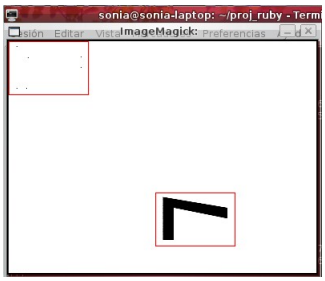

Figure 4.7: Example of translation. Frame 105

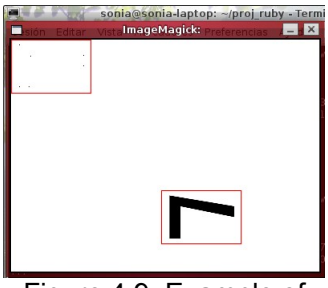

Figure 4.9: Example of translation. Frame 107

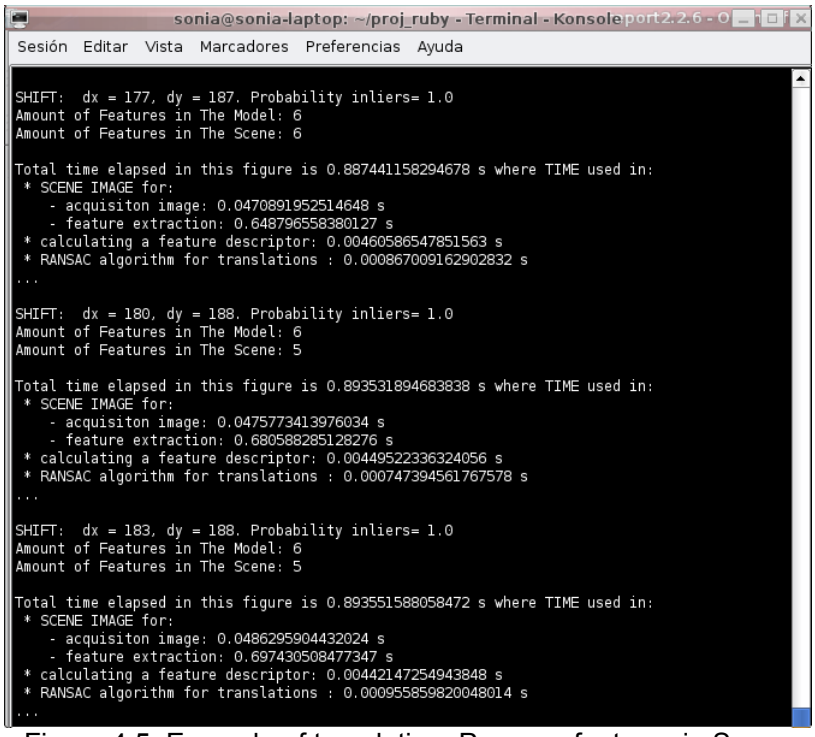

Figure 4.5: Example of translation. Program features in Scene Frames [103, 105]

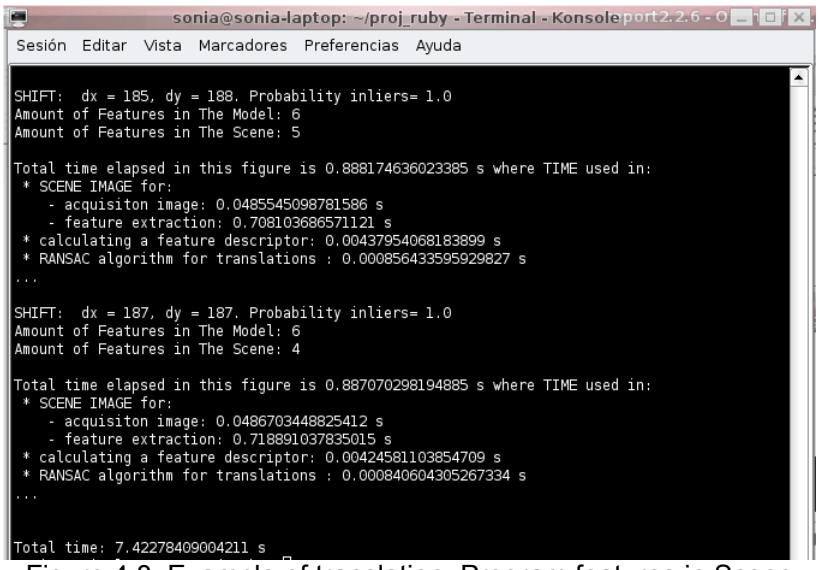

Figure 4.8: Example of translation. Program features in Scene Frames [106, 107]

As one can see, the major part of the execution time is spent on feature extraction. Execution time of RANSAC algorithm is the smallest in comparison of the rest of significant sections.

4.1.2 Example 2: Rotation-RANSAC

In this part, some results of object recognition with three degrees of freedom are shown. Tests were made over a video where the object underwent translation and rotation. The software was executed three times for each case. The next table shows the same variables than the translation one. The results were obtained in a similar example but using other different frames.

Using an algorithm where the maximum number of times is 200 (T = 200) and a threshold of probability of finding inliers is 80%, the next results were obtained:

Times	Total Frames	Good Results	% Good Results
1	248	114	45.97%
2	248	114	45.97%
3	248	114	45.97%

As you can observe from the table, results obtained were worse than in translation case, since only almost a half of frames reached to recognise the object.

In the following example, the images shown different frames in the tracking of rotation in a figure (in this case it is shown from frame 178 to frame 185). The model figure is recognised in each frame, although the position is different to the initial one as well (as in displacement section).

In these figure, a red frame is also drawn by means of the model features, translation and rotation matrices. And the red line joins the two selected randomly features.

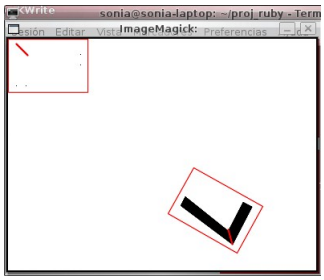

Figure 4.10: Example of rotation. Frame 178

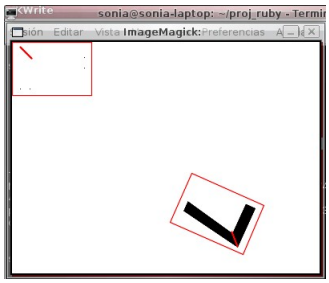

Figure 4.12: Example of rotation. Frame 179

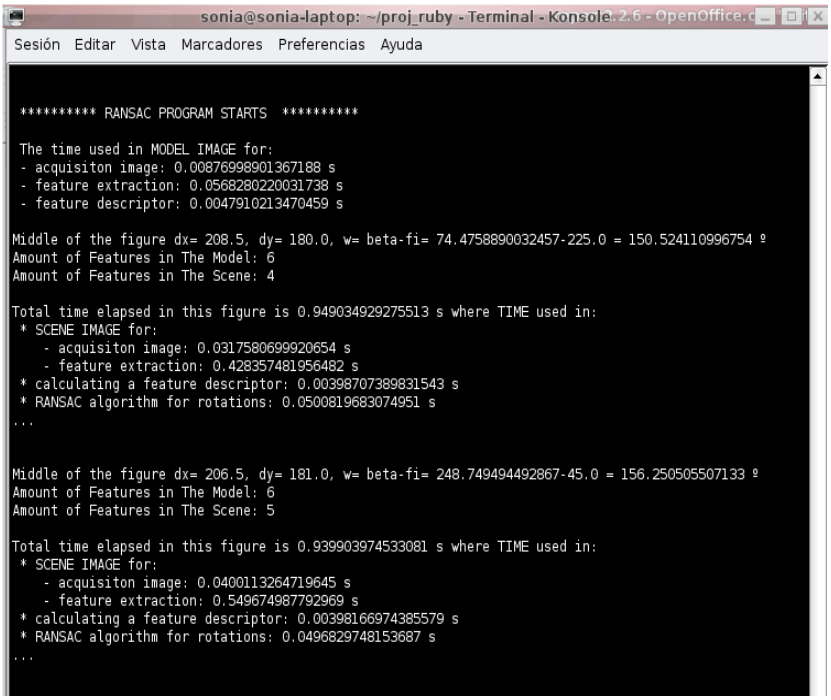

Figure 4.11: Example of rotation. Program features in Model and Scene Frames [178, 179]

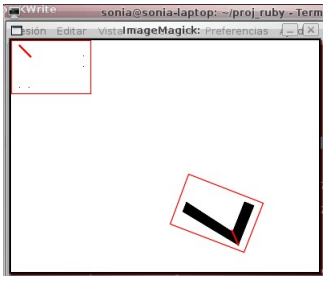

Figure 4.13: Example of rotation. Frame 181

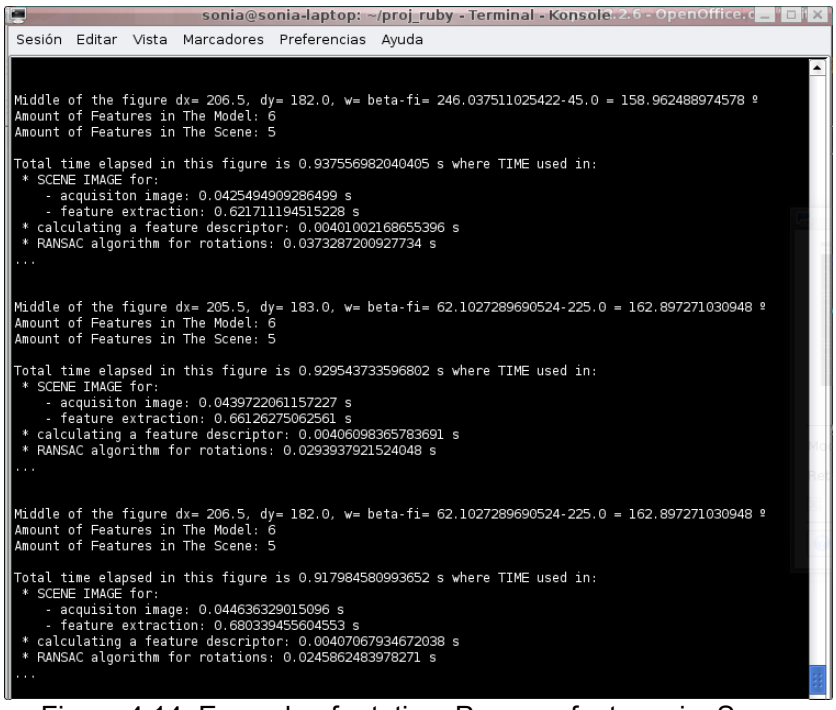

Figure 4.14: Example of rotation. Program features in Scene Frames [180, 182]

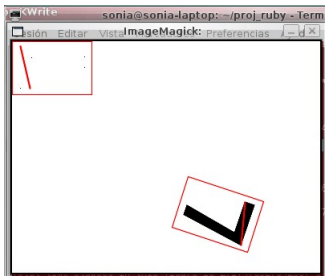

Figure 4.15: Example of rotation. Frame 183

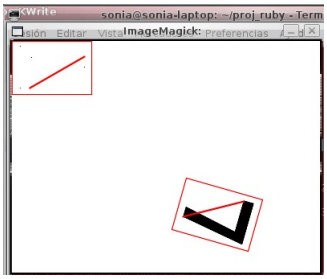

Figure 4.17: Example of rotation. Frame 184

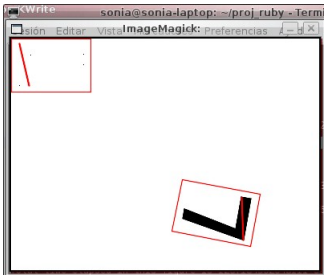

Figure 4.18: Example of rotation. Frame 185

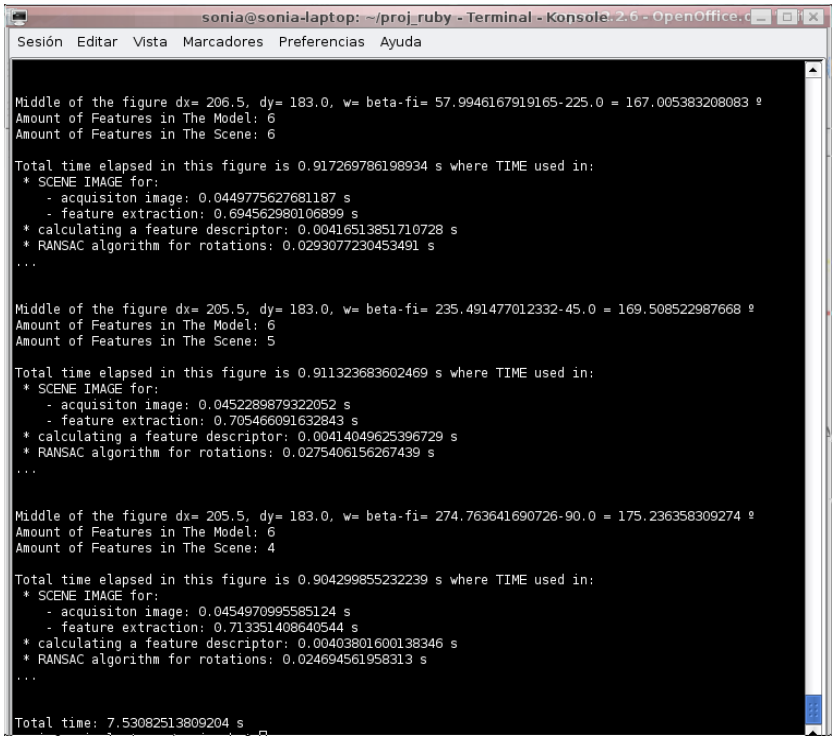

Figure 4.16: Example of rotation. Program features in Scene Frames [183, 185]

In rotation case, the execution time of RANSAC algorithm is bigger than RANSAC for translations. However, the major part of the execution time is used for feature extraction.

Note that all examples change in each time that the program is run, due to RANSAC is a random method. Therefore, the results are not always the same.

4.2 EXECUTION EXAMPLES IN REAL DATA

In this section, an example in real data will be shown. This example consists in recognition of a book, see Figure 4.19. This book has undergone translation and rotation motions in a plane. Firstly, several tables about results in this example video will be shown where the threshold of the probability was changed. Secondly, some frames of the video in motion will be illustrated.

Figure 4.19: Example of
real data. Model Image

Thus, several executions of the program will be made where some parameters had to be modified. In the following tables, it can be seen the different results which were obtained by modifying the threshold of probability in good points. Note that *Good Results* means frames where the object is recognised and *% Good Results* a rate of object recognition in this example.

- Threshold of probability of finding inliers: 40%

Times	Total Frames	Good Results	% Good Results
1	254	163	64.17%
2	254	170	66.93%
3	254	175	68.90%

- Threshold of probability of finding inliers: 60%

Times	Total Frames	Good Results	% Good Results
1	254	65	25.59%
2	254	67	26.38%
3	254	64	25.20%

- Threshold of probability of finding inliers: 70%

Times	Total Frames	Good Results	% Good Results
1	254	13	5.12%
2	254	15	5.91%
3	254	17	6.69%

As one can observe in these results, the better choice is a threshold of 40%. However, choose this low probability means that sometimes the recognised object will not be the searched one.

A real data example appears in the Figure 4.20. You can observe how the book (Figure 4.19) is recognised by means of the corners of the model image. The different illustrated frames were obtained using a minimum inliers probability of 40%.

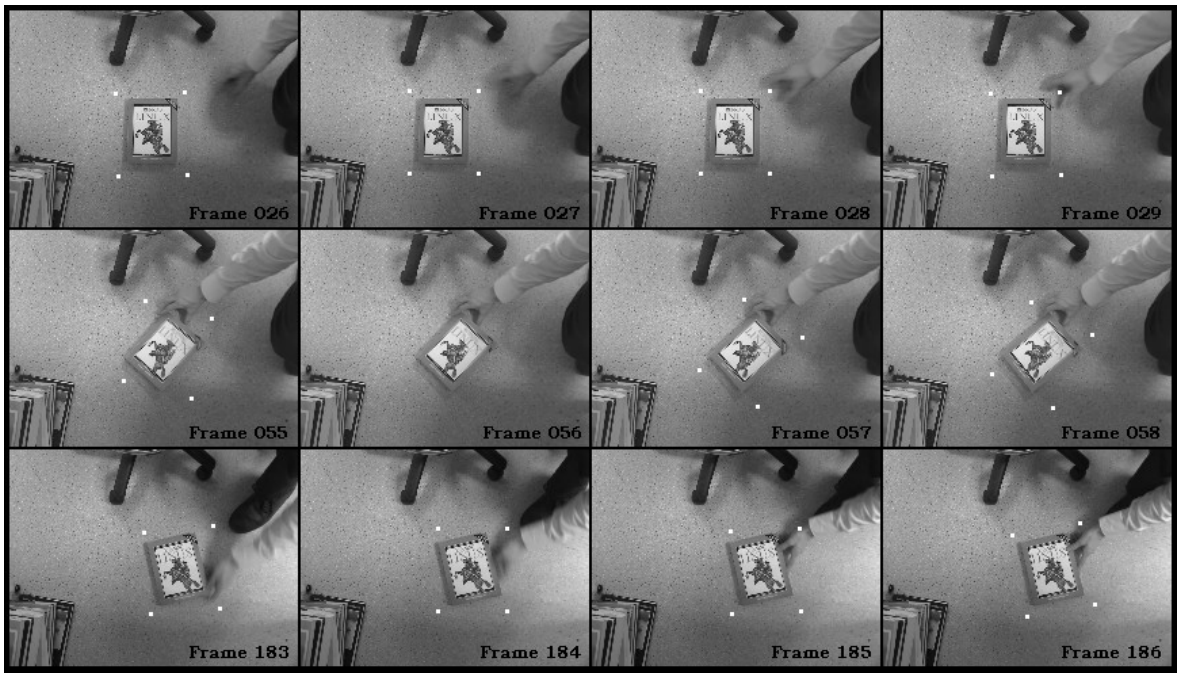

Figure 4.20: Example of real data. Rotation and Translation

Chapter 5.

Conclusions and Future Works

Real-time recognition of rigid object can be applied to various fields, specifically if it is applied to microscopic objects, it can analyse particles, for example, in medical field or it can be used in industry for automating the process of assembly for small pieces.

To conclude, this project has achieved to recognise 2D rigid object in simulated data. Thus, feature extraction algorithm, feature descriptor part and RANSAC algorithm perform well in the three degree problems (translations in X-direction and in Y-directions and rotations in 2D). However, when the software was applied to real data, results were worse and some parameters of RANSAC had to be modified. Furthermore, it was observed that the feature descriptor could be changed in order to improve the software performance.

After evaluating the execution time of different sections of algorithm, it was observed that RANSAC speeds up this proccess of recognition.

Possible improvements and future work are:

- Reduce the execution time by changing the feature extraction algorithm.

- Test different parameters, because when the algorithm was applied to real objects, the algorithm did not perform well.

- Unify RANSAC algorithm by using matrices in the displacement as well. However, this method may increase the execution time.

- Extend RANSAC implementation to four degrees of freedom problems, *i.e.*, to displacement in 3D.

- Change the feature descriptor by using appereance templates and the normalised cross correlation instead of grey-level histograms [12].

- Do not use non-maxima suppression in order to achieve real-time performance [12].

Appendix I. Main code

In this appendix is shown the main source code of the implemented software in this project work.

```
##############################################################################################
###                                  RANSAC PROJECT                                        ###
###    Name:          RECOGNITION OF 2D-OBJECTS USING RANSAC.                              ###
###    Description:   Recognition of an object which is in a different                     ###
###                   position in the plane than an original model.                        ###
###                   This recognition is made by means of RANdom SAmple                   ###
###                   Consensus (RANSAC).                                                  ###
###    Author:        Sonia Fernández Rodríguez                                            ###
###                   Sheffield Hallam University                                          ###
###    Last update:   20/05/2008                                                           ###
###                                                                                        ###
##############################################################################################


require 'hornetseye'
require 'RMagick'

# This matrix is neccessary to calculate correctly the reverse matrix
require 'matrix_fix'

# To calculate the time of the parts in a way easier.
require 'rtimer'

include Hornetseye
include Math


#-------------------------------------------------------------------------------------------------#
#                                        METHOD DEFINITIONS                                       #
#-------------------------------------------------------------------------------------------------#

# Calculate a Histogram of several points (features) of an image (image) normalised in the range
# [0.0, 15.0] by considering around these points an extend of (width)
def calculateHistogram( features, image, width )
  histogram = {}
  features.each do |x,y|
    histogram[[x,y]] = [0]*16
    image[x-width..x+width, y-width..y+width].each do |p|
        histogram[[x,y]][p] += 1
    end
  end
    return histogram
end
```

```
# Build the matrix form to do a 2D affine transformations, in this case
# combine translation and rotation.
def iso2d( w, t1, t2 )
  Matrix[[Math::cos(w),-Math::sin(w),t1],[Math::sin(w),Math::cos(w),t2],[0,0,1]]
end

def xramp( *shape )
  retval = MultiArray.new( MultiArray::LINT, *shape )
  for x in 0...shape[0]
    retval[ x, 0...shape[1] ] = x
  end
  retval
end

def yramp( *shape )
  retval = MultiArray.new( MultiArray::LINT, *shape )
  for y in 0...shape[1]
    retval[ 0...shape[0], y ] = y
  end
  retval
end

class Object

  # Coppy an object in a file and return it
  def dclone
     Marshal.load(Marshal.dump(self))..
  end

end

class MultiArray

  # Kanade-Lucas-Tomasi coner-detector
  def klt( sigma = 1.0 )

     x, y = gauss_gradient_x( sigma ), gauss_gradient_y( sigma )

     a = ( x * x ).gauss_blur( sigma )
     b = ( y * y ).gauss_blur( sigma )
     c = ( x * y  ).gauss_blur( sigma )

     tr = a + b
     det = a * b - c * c
     dissqrt = ( tr * tr - det * 4 ).clip_lower( 0.0 ).sqrt
     features = 0.5 * ( tr - dissqrt )
     return features

  end

  # Make Non-Maxima-Suppression to limit the number of features
  def nms( threshold, fringe = 1 )
    box = [ fringe...(shape[0]-fringe), fringe...(shape[1]-fringe) ]
    mask = self[ *box ].binarise( threshold )
    xr, yr = xramp( *mask.shape ) + fringe, yramp( *mask.shape ) + fringe
    retval = []
    valMask = MultiArray.to_multiarray( [ [ 1, 1, 1 ], [ 1, 0, 1 ], [ 1, 1, 1 ] ] ).
              to_type( MultiArray::UBYTE )
    xr.mask( mask ).multi_each( yr.mask( mask) ) do |x,y|
        if self[ x,y ] > self[ (x-1)..(x+1), (y-1)..(y+1) ].mask( valMask ).max
            retval.push( [ x, y ] )
        end
    end
    return retval
  end
```

```
  # Return the original image with the choosen features in white
  def feature_image( features, fringe = 1 )
    result = dup
    features.each do |x,y|
      if x > 1 and y > 1 and x < (shape[0]-1) and y < (shape[1]-1)
        result[ (x-fringe)..(x+fringe), (y-fringe)..(y+fringe) ] = 255
      end
    end
    result
  end

end

class Array

  # Calculate the sum total of all elements belonging to a vector
  def sum
    inject(0){ |v,i| v+i }
  end

  # Calculate the correlation coefficient of two arrays
  def correlation_coef( other )
    pairs = zip( other )
    sumx = self.sum
    sumy = other.sum
    sumxx = ( collect { |x| x**2 } ).sum
    sumyy = ( other.collect { |y| y**2 } ).sum
    sumxy = ( pairs.collect { |xy| xy[0] * xy[1] } ).sum
    n = size
    # Pearson correlation coefficient
    ( n*sumxy - sumx*sumy )/sqrt(( n*sumxx - sumx*sumx ) * ( n*sumyy - sumy*sumy ))
  end

  # Find the best match in funtion of the distance between two points of the model and the scene image
  def findMatch(td, auxCoef, p1, p2, dm )
    aux1 = []
    aux2 = []
    d = 0
    if !auxCoef[self[0]].empty?

      aux1 = auxCoef[self[0]].pop
      aux1 = aux1[0]

      d = ((p2[0]-aux1[1][0])**2 + (p2[1]-aux1[1])**2)**0.5

      # If the distance difference with the new point, aux1, is less than the threshold, being this new
      distance.
      # different of 0, or the distance diffence is less than the previous diffence, being this new
      distance.
      # different of 0, this new point will be considered as "good point" being this new distance
      different of 0.
      if ( (d-self[2]).abs <= td and d != 0 ) or ( (d-self[2]).abs < (dm-self[2]).abs and d != 0 )
        p1 = aux1
        dm = d
      end
    end

    if !auxCoef[self[1]].empty?

      aux2 = auxCoef[self[1]].pop
      aux2 = aux2[0]

      d = ((aux2[0]-p1[0])**2 + (aux2[1]-p1[1])**2)**0.5

      if ( (d-self[2]).abs <= td and d != 0 ) or ((d-self[2]).abs < (dm-self[2]).abs and d != 0 )
        p2 = aux2
        dm = d
      end
    end
```

```
    if !aux1.empty? and !aux2.empty?

      d = ((aux2[0]-aux1[0])**2 + (aux2[1]-aux1[1])**2)**0.5
      if d != 0 and  (d-self[2]).abs < (dm-self[2]).abs
        p1 = aux1
        p2 = aux2
        dm = d
      end

    end
    return auxCoef, p1, p2, dm

  end

  # Calculate the 2D coordenates of a scene point from a model point by using the rotation matrices of
  # the scene image and the rotation image
  def to_scnPoint( rotScn, rotMod)
    modPoint = Vector[self[0],self[1],1]
    scnPoint = rotScn * rotMod.inv * modPoint
    coorScn = [scnPoint[0].round, scnPoint[1].round]
    return coorScn
  end


  # Calculate the 2D coordenates of a model point from a scene point by using the rotation matrices of
  # the scene image and the rotation image
  def to_modPoint( rotMod, rotScn)
    scnPoint = Vector[self[0],self[1],1]
    modPoint = rotMod * rotScn.inv * scnPoint
    coorMod = [modPoint[0].round, modPoint[1].round]
    return coorMod
  end

end

#-------------------------------------------------------------------------------------------------------#
#                                        END METHOD DEFINITIONS                                         #
#-------------------------------------------------------------------------------------------------------#


executionTimes = { "MAIN_PROGRAM" => Timer.new, "EACH_SCENE" => Timer.new ,"ACQUISITION" => Timer.new,
"EXTRACTION" => Timer.new, "DESCRIPTOR" => Timer.new, "RANSAC_SHIFT" => Timer.new, "RANSAC_ROT" =>
Timer.new }
rotation = true   # The variable "rotation" indicates in the image motion if the orientation changes,
                  #  where rotation = true, or only the image position changes, where rotation = false
r = 0
goodresults = 0   # This variable account the number of recognised objects
width = 3
histogramMod = {}
executionTimes["MAIN_PROGRAM"].start

#---------------------------------------- MODEL IMAGE ----------------------------------------------#


if $0 == __FILE__
  video = XineInput.new( "runninglinux.avi" )
  fringe = 2; sampling = 3; bins = 6
  x0 = 240/sampling; x1 = (240+151)/sampling;
  y0 = 179/sampling; y1 = (179+179)/sampling
  # Load a model image
  executionTimes["ACQUISITION"].start
  2.times { video.read }
  video_image = video.read_grey8.
    downsample( [ sampling, sampling ], [ sampling / 3, sampling / 3 ] )
  imgModel = video_image[ x0..x1, y0..y1 ]
  executionTimes["ACQUISITION"].stop
```

```
  # Localitation of features from model image
  executionTimes["EXTRACTION"].start
  model = imgModel.klt

  threshold = model.max/3.0   # Limit the seek of features by taking only the point bigger than 1/3
maximum pixel of model
  featuresMod = model.nms( threshold, width )   # Obtaining of model features

  executionTimes["EXTRACTION"].stop

  display = X11Display.new
  template_output = OpenGLOutput.new
  template_window = X11Window.new( display, template_output,
                                   model.shape[0] * sampling,
                                   model.shape[1] * sampling )
  template_window.title = "Template"
  template_output.write( imgModel.feature_image( featuresMod ) )
  template_window.show


  # Normalise the image's pixels between 0 and 15
  newModel = imgModel.normalise(0.0..15.0).to_type(MultiArray::LINT)

  executionTimes["DESCRIPTOR"].start
  histogramMod = calculateHistogram( featuresMod, newModel, width )
  executionTimes["DESCRIPTOR"].stop


  puts "\n\n ********** RANSAC PROGRAM STARTS  ********** \n"
  puts "\n The time used in MODEL IMAGE for: \n - acquisiton image: " +
executionTimes["ACQUISITION"].inspect
  puts " - feature extraction: " + executionTimes["EXTRACTION"].inspect
  puts " - feature descriptor: " + executionTimes["DESCRIPTOR"].inspect


#------------------------------------ END MODEL IMAGE ----------------------------------------#

  # Display a video in a window with the model image and with this dimension: 320 x 200
  video_output = XVideoOutput.new
  video_output = OpenGLOutput.new
  video_window = X11Window.new( display, video_output,
                                video_image.shape[0] * sampling,
                                video_image.shape[1] * sampling )
  video_window.title = "Video RANSAC"
  video_window.show

  #outvideo = MEncoderOutput.new( "realDataRotation1.avi", 25 )


  if not featuresMod.empty?
    quit = false

    # Loop to load several scene images
    while video.status? and video_output.status? and ( not quit )
      #---------------------------------- SCENE IMAGE ----------------------------------------#
      executionTimes["EACH_SCENE"].start
      featuresScn = []
      r += 1

      # Localitation of features from scene image
      executionTimes["EXTRACTION"].start
      scene = video_image.klt
      featuresScn = scene.nms( threshold, width )...
      executionTimes["EXTRACTION"].stop
```

```
if not featuresScn.empty?
  histogramScn = {}
  thresholdRansac = 6

  # Normalise the image's pixels between 0 and 15
  newScene = video_image.normalise(0.0..15.0).to_type(MultiArray::LINT)


  #---------------------------------- END_SCENE IMAGE ---------------------------------------#

  # ------ FEATURE DESCRIPTOR ------- #

  executionTimes["DESCRIPTOR"].start

  histogramScn = calculateHistogram( featuresScn, newScene, width )
  executionTimes["DESCRIPTOR"].stop


  # ------ END_FEATURE DESCRIPTOR ------- #

  # Calculate the correlation coefficients between the Scene and the Model pixels

  coefScnMod = {}
  matches = {}


  featuresScn.each do |xs,ys|
    aux = []
    featuresMod.each do |xm,ym|
      rxy = histogramScn[[xs,ys]].correlation_coef( histogramMod[[xm,ym]] )
      # if rxy > 0.5  # Only select the features that have a relationship
      aux.push( [[xm,ym],rxy] )
      # end
    end
    # Sorted the correlation coefficient in ascending order
    coefScnMod[[xs,ys]] = aux.sort_by{ |coefficient| coefficient[1] }
  end


  if not rotation

    #---------------------------------- TRANSLATION ---------------------------------------#

    tp = 0.9 # Minimum probability to have a good match
    pi = 0   # Probability calulated for the inlier pixels
    dx = 0   # Coordinate X of the motion vector
    dy = 0   # Coordinate Y of the motion vector
    times = 0

    #------------------- SHIFT-RANSAC ----------------------#
    executionTimes["RANSAC_SHIFT"].start

    while pi <= tp and times < 200

      #  xScene = featuresScn[position][0]
      #  yScene = featuresScn[position][1]
      #  xModel = matches[[xScene,yScene]][0][0]
      #  yModel = matches[[xScene,yScene]][0][1]

      inliers = 0
      # A scene feature is randomly chosen
      position = rand(featuresScn.size)

      # Calculate the motion vector for this feature
      matches[featuresScn[position]] = coefScnMod[featuresScn[position]].last
      dx = featuresScn[position][0] - matches[featuresScn[position]][0][0]
      dy = featuresScn[position][1] - matches[featuresScn[position]][0][1]
```

```
    # Consensus
    featuresMod.each do |xm, ym|
      callcc do |c|
        featuresScn.each do |xs, ys|
          distance = ((xm+dx - xs)**2 + (ym+dy - ys)**2)**0.5
          if distance <= thresholdRansac
            inliers += 1
            c.call
          end
        end
      end
    end

    # if featuresScn.size < featuresMod.size
    pi = inliers.to_f/featuresScn.size
    # else
    #   pi = inliers.to_f/featuresMod.size
    # end

    times += 1

  end
  executionTimes["RANSAC_SHIFT"].stop
  #----------------- END_SHIFT-RANSAC --------------------#


  puts "SHIFT:  dx = #{dx}, dy = #{dy}. Probability inliers= #{pi}\n"

  #------------------------------- END_TRANSLATION ----------------------------------#

else

#-------------------------------- ROTATION ----------------------------------------#

  thresholdDist = 4  # Minimum error between the difference of distances between points of
scene and model
  tp = 0.4           # Minimum probability to have a good match
  pin = 0            # Probability calulated for the inlier pixels

  p1 = []            # Points of model image
  p2 = []

  ps1 = []           # Points of scene image
  ps2 = []

  dm = 0             # Distance between the two points chosen from model image
  ds = 0             # Distance between the two points chosen from scene image

  omega = 0          # Rotation-angle of the figure
  dx = 0             # Coordinate X of the motion vector
  dy = 0             # Coordinate Y of the motion vector

  times = 0

  #------------------- ROTATION-RANSAC ----------------------#
  executionTimes["RANSAC_ROT"].start

  while pin < tp and times < 100

    choosenFeatures=[]

    # Select randomly 2 different features from the Scene image
    ps1 = featuresScn[rand(featuresScn.size)]
    ps2 = featuresScn[rand(featuresScn.size)]

    while ps2 == ps1
      ps2 = featuresScn[rand(featuresScn.size)]
    end
```

```
if !coefScnMod[ps1].empty? and !coefScnMod[ps2].empty?
  # Take the suitable features from coefScnMod for each scene point selected
  p1 = coefScnMod[ps1].last[0]
  p2 = coefScnMod[ps2].last[0]

  # Calculate the both distances between the 2 points
  ds = ((ps2[0]-ps1[0])**2 + (ps2[1]-ps1[1])**2)**0.5
  dm = ((p2[0]-p1[0])**2 + (p2[1]-p1[1])**2)**0.5


  # To solve when dm-ds is not minus than thresholdDist, a better solution is searched 3
times at the most
  j = 0
  while (dm-ds).abs >= thresholdDist and j < 3 and !coefScnMod[ps1].empty? and
!coefScnMod[ps2].empty?

    p1 = coefScnMod[ps1].last[0]
    p2 = coefScnMod[ps2].last[0]

    dm = ((p2[0]-p1[0])**2 + (p2[1]-p1[1])**2)**0.5
    j += 1

  end


  if (dm-ds).abs < thresholdDist

    t = []
    tm = []
    inliers = 0

    ### Calculation of the rotation angle  ###

    dxs = ps2[0]-ps1[0]
    dys = ps2[1]-ps1[1]

    beta = atan2(dys,dxs)       # beta is the angle of the scene image's vector

    dxm = p2[0]-p1[0]
    dym = p2[1]-p1[1]

    fi = atan2(dym,dxm)         # fi is the angle of the model image's vector

    # omega = beta - fi         # omega is the angle of the rotation of the figure

    tm = [(p2[0]+p1[0])/2.0,(p2[1]+p1[1])/2.0 ]
    t = [(ps2[0]+ps1[0])/2.0,(ps2[1]+ps1[1])/2.0 ]

    rotScn = iso2d(beta,t[0],t[1])
    rotMod = iso2d(fi,tm[0],tm[1])

    # Consensus

    featuresMod.each do |xm, ym|
      callcc do |c|
        coorScn = [xm,ym].to_scnPoint( rotScn, rotMod )
        featuresScn.each do |xs, ys|
          distance = ((coorScn[0] - xs)**2 + (coorScn[1] - ys)**2)**0.5
          if distance <= thresholdRansac
            inliers += 1
            choosenFeatures.push([xm,ym])
            c.call
          end
        end
      end
    end
```

```
            pin = inliers.to_f/featuresScn.size

          end

          times += 1

        end
      end

      executionTimes["RANSAC_ROT"].stop
      #------------------- END_ROTATION-RANSAC ----------------------#


      result = video_image.to_hornetseye.to_rgb24.to_magick

      framePoints = []

      if pin > tp

        goodresults +=1
        #--------- The inlier probability reachs the threshold of minimal probability -----#
        coorPoint1=[0,0]
        coorPoint2=[imgModel.shape[0],0]
        coorPoint3=[0, imgModel.shape[1]]
        coorPoint4=[imgModel.shape[0],imgModel.shape[1]]

        coorScn1 = coorPoint1.to_scnPoint( rotScn, rotMod )
        coorScn2 = coorPoint2.to_scnPoint( rotScn, rotMod )
        coorScn3 = coorPoint3.to_scnPoint( rotScn, rotMod )
        coorScn4 = coorPoint4.to_scnPoint( rotScn, rotMod )

        framePoints=[coorScn1,coorScn2,coorScn3,coorScn4]

        #output.write( result.to_hornetseye.to_yv12 )
        # display.processEvents
        puts "..."

      end

      #------------------------------- END_ROTATION ---------------------------------------#

    end
  end
  # End of featuresScn.empty?

  executionTimes["EACH_SCENE"].stop

  print "Amount of Features in The Model: #{featuresMod.size}\n"
  print "Amount of Features in The Scene: #{featuresScn.size}\n"

  puts " \nTotal time elapsed in this figure is " + executionTimes["EACH_SCENE"].inspect + " where
TIME used in: "
  puts " * SCENE IMAGE for: \n    - acquisiton image: " + executionTimes["ACQUISITION"].inspect
  puts "    - feature extraction: " + executionTimes["EXTRACTION"].inspect
  puts " * calculating a feature descriptor: " + executionTimes["DESCRIPTOR"].inspect

  if not rotation
    puts " * RANSAC algorithm for translations : " + executionTimes["RANSAC_SHIFT"].inspect
  else
    puts " * RANSAC algorithm for rotations: " + executionTimes["RANSAC_ROT"].inspect
  end

  puts "..."
  puts "\n"
```

```
      template_output.write( imgModel.feature_image( choosenFeatures ) )
      video_output.write( video_image.feature_image( framePoints ) )
      #outvideo.write( video_image )
      display.processEvents

      begin
        # Load a scene image
        video_image = video.read_grey8.
          downsample( [ sampling, sampling ], [ sampling / 3, sampling / 3 ] )
      rescue RuntimeError => e
        quit = true
      end
      # 3D image: x,y=spacial coordinates, z=histogram

    end
    # End-Loop to load several scene images

    executionTimes["MAIN_PROGRAM"].stop

    puts "\nTotal time: " + executionTimes["MAIN_PROGRAM"].inspect


  else
    template_window.title = "***"
    video_window.title = "***"
  end
  # end of featuresMod.empty?
  puts "FINAL: R times: #{r}; GOOD RESULTS: #{goodresults}"
end
```

Appendix II. Matrix Class code

In the program was necessary use a complementary function of the Matrix Class because there was a problem in the execution of inverse matrix in ruby. Thus, it was necessary use 'matrix_fix'. This file has the following code.

```
require 'matrix'
class Matrix
  def inverse_from(src)
    size = row_size - 1
    a = src.to_a

    for k in 0..size

      i = k
      akk = a[k][k].abs
      for j in (k+1)..size
        v = a[j][k].abs
        if v > akk
          i = j
          akk = v
        end
      end
      Matrix.Raise ErrNotRegular if akk == 0
      if i != k
        a[i], a[k] = a[k], a[i]
        @rows[i], @rows[k] = @rows[k], @rows[i]
      end
      akk = a[k][k]

      for i in 0 .. size
        next if i == k
        q = a[i][k] / akk
        a[i][k] = 0

        (k + 1).upto(size) do
          |j|
          a[i][j] -= a[k][j] * q
        end
        0.upto(size) do
          |j|
          @rows[i][j] -= @rows[k][j] * q
        end
      end

      (k + 1).upto(size) do
        |j|
        a[k][j] /= akk
      end
```

```
        0.upto(size) do
          |j|
          @rows[k][j] /= akk
        end
      end
      self
    end
  end
```

Appendix III. Timer Class Code

A new class were implemented in order to count the amount of time employed in each part of an algorithm in a way easier. This new class is Timer and it is shown in the next code.

```
class Timer
  def initialize
    @c = 0
    @t = 0
  end
  def start
    @t -= Time.new.to_f
    self
  end
  def stop
    @t += Time.new.to_f
    @c += 1
    self
  end
  def value
    @t / @c
  end
  def inspect
    if @c > 0
      if @t >= 0
        "#{@t / @c} s"
      else
        "#{(@t+Time.new.to_f)/@c}"
      end
    else
      "0 s"
    end
  end
end
```

Bibliography

Websites

[1] Main Page- MMVL Wiki. http://vision.eng.shu.ac.uk/mmvlwiki/index.php/Main_Page Last accessed April 2nd, 2008.

[2] Microsystems and Machine Vision Laboratory | Sheffield Hallam University. http://www.shu.ac.uk/mmvl/ Last accessed: April 2nd, 2008.

[4] Jan Wedekind. HornetsEye – Computer Vision for the Robotic Age. http://www.wedesoft.demon.co.uk/hornetseye-api/ Last accessed: April 3rd, 2008.

[5] Stan Birchfield. KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker. Website. http://www.ces.clemson.edu/~stb/klt/ Last accessed: April 4th, 2008.

[9] CSC320S Schedule & Notes. http://www.cs.toronto.edu/~kyros/courses/320/Lectures.s07/lecture.2007s.08.pdf/ Last accessed: October 15th, 2007.

[10] Ruby (programming language) – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Ruby_%28programming_language%29 Last accessed: April 20th, 2008.

[13] Patente: PROCEDIMIENTO DE ESTIMACION DEL MOVIMIENTO ENTRE DOS IMAGENES CON GESTION DE LAS INVERSIONES DE MALLAS Y PROCEDIMIENTO DE CODIFICACION CORRESPONDIENTE. http://www.invenia.es/oepm:e00988902/ Last accessed: January 22nd, 2008.

[14] Transformaciones. Universidad de Navarra. http://www.tecnun.es/asignaturas/grafcomp/presentaciones/transformaciones.ppt Last accessed: April 14th, 2008.

Books

[6] D. Forsyth and J. Ponce. *Computer Vision – A Modern Approach*. Prentice Hall, 2003.

Articles

[3] Chu-Song Chen, Yi-Ping Hung and Jen-Bo Cheng. *RANSAC-Based DARCES: A New Approach to Fast Automatic Registration of Partially Overlapping Range Images*. In IEEE Transactions on pattern analysis and machine intelligence, Nov. 1999, volume 21, pages 1229-1234. Acad. Sinica, Taipei, Taiwan.

[7] M. Zuliani, C.S. Kenney, and B.S. Manjunath. *The MultiRansac algorithm and its application to detect planar homographies*. In ICIP 2005: IEEE International Conference on Image Processing, 11-14 Sept. 2005, volume 3, pages III - 153-6. California Univ., Santa Barbara, CA, USA.

[8] M.A. Fischler and R.C. Bolles. *Random Sample Consensus: A Paradighm for Model Fitting with Applications to Image Analysis and Automated Cartography*. June 1981, volume 24, number 6. SRI International.

[11] Jianbo Shi and Carlo Tomasi. *Good features to track*. In Proceedings of the 1994 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Jun 21-23 1994, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pages 593–600, Seattle,WA, USA, 1994. Cornell Univ, Ithaca, NY, USA, Publ by IEEE, Los Alamitos, CA, USA.

[12] Mark Lloyd Pupilli. *Particle Filtering for Real-time Camera Localisation*.PhD thesis, University of Bristol, 2006.