Obstacle Detection for Mobile Robots Using Computer Vision

Syedur Rahman

Final Year Project (PR3)
Supervised by Dr N.E. Pears

Department of Computer Science
University of York

March 2005

Word Count: 22,096 words excluding the appendix
Words were counted using word count on MS Word

Abstract

Computer vision is a field of computer science that has been heavily researched in recent years. Its applications in robotics are diverse ranging from face recognition to autonomous navigation. This project aims to research one of computer vision's most important contributions to the navigation of mobile robots - obstacle detection

Multi-view relations are used as the fundamental principles upon which the solution is based. Planar homography, epipolar geometry and image segmentation are used to analyse the reliability of three obstacle detection methods comparing and contrasting their performance in different scenes.

Pairs of images of a scene are taken by a digital camera that is moved between images. The first method uses planar homography to create a warped image from the initial image and performs obstacle detection via its comparison to the final image. The second method estimates heights of corners on the scene using planar homography and gives the heights of each distinguishable object on the scene using image segmentation. The third method uses epipolar geometry and edge detection to find point correspondences on edges between the images and then uses planar homography to compute the heights along the contours thereby performing obstacle detection.

Contents

List of Figures

List of Tables

1 INTRODUCTION

Recently quite a significant amount of time and resources spent in the field of computing has been directed into computer vision. What has drawn so much interest into the subfield is perhaps the inexplicable ease and immaculate accuracy with which the human brain accomplishes certain tasks related to vision. The concept that the human brain is essentially an information processing device itself, much like the modern computer, is what computer scientists have based much of their ideas and inspirations on when researching into computer vision.

Although for a long time biologists have delved into and unravelled a proportion of the mysteries of the brain, we are yet to be capable of mimicking its functionality with the aid of a machine. This project aims to research into one of the human brain's most important image processing operations to be accomplished by a computer – obstacle detection. It provides some insight into several basic techniques used in fields of computing such as image processing, pattern recognition and computer vision that can be put together with a few renowned and a few innovative approaches to perform basic obstacle detection for a machine.

1.1 Obstacle Detection and Previous work

Obstacle detection is defined as "the determination of whether a given space is free of obstacles for safe travel by an autonomous vehicle" by Singh [20]. Obstacle detection is one of the most renowned problems within the subfield of computer vision in terms of the amount of research it has attracted and the number of uses it has. Together with research into other subfields of artificial intelligence, obstacle detection is crucial in order to perform many basic operations for mobile robots such as avoidance and navigation.

A good obstacle detection system must be capable of the following [Singh]:

- to detect obstacles on a given space in good time
- to detect and identify correct obstacles
- to identify and ignore ground features that may appear as obstacles

Previous work into this field has made use of digital cameras, laser scanners, sonar, odometry etc. to perform obstacle detection. "Snap shots" are taken of the real scene using these input devices and the data is processed by a computer which ultimately performs obstacle detection. Since most of these devices can only take a limited number of shots of a scene in a given time and since an enormous amount of data needs to be processed, obstacle detection can often be quite slow for some real time applications, e.g. navigation of high-speed vehicles. Very sophisticated sensors and processors used to perform obstacle detection so far have been able to accommodate navigation at speeds of only a few metres per second.

1.2 This Project

Although several other methods such as odometry, infrared and laser have been used to achieve obstacle detection, this project uses vision, which is the processing of images dependent on light radiated from the robot's environment to attempt to build a reliable obstacle detection system.

Despite the final aim being the construction of a system, more emphasis is put on research than engineering throughout this project. A comprehensive study is made of techniques from several fields of computer science that contribute to obstacle detection before the system is implemented. The product of this project is in fact three obstacle detection systems rather than one each using a sub-set of these techniques. These are:

- Obstacle detection using planar homography and image warping (via comparison to warped image)
- Obstacle detection using planar homography and image segmentation (via computation of heights of segments)
- Obstacle detection using epipolar geometry, planar homography and edge detection (via computation of heights along contours)

The ultimate goal of this project is in fact the evaluation of each of these "sub-systems" in different operating environments or scenes, finally comparing and contrasting their performances. The construction of a system that detects obstacles in real time is not attempted. Rather, the aim is to construct and evaluate a system that detects obstacles reliably.

1.3 Overview of report structure

The report for this project is structured into six chapters. Chapter 1 has so far been an introduction to obstacle detection and to the project itself. The structure of the remainder of this project is as follows:

Chapter 2 entitled "Literature review" briefly summaries all the background reading done to gain enough insight into computer vision to implement this project. It includes a brief history of computer vision followed by descriptions of the basic concepts within the sub-field and detailed explanations of their applications in obstacle detection.

Chapter 3 entitled "Design" consists of a more comprehensive list of requirements followed by a discussion on the selection of design methodoly appropriate for it. It also includes details on the given hardware constraints, followed by a comparison between software development environments applicable for this project justifying the selection of the most appropriate one. This is followed by a design plan which is basically the identification and description of each sub-problem within the system and the relationships between them.

Chapter 4 entitled "Implementation" comprehensively describes the solutions to each of the sub-problems identified during design and their integration into the final intended system. Descriptions of the solutions to sub-problems are made more vivid with the help of diagrams and pseudocode.

Chapter 5 entitled "Evaluation" starts off with the description of a comprehensive test plan for the system that would examine its functionality on a broad range of scenarios. Then a more detailed description of the tests conducted and results achieved for each of the major sub-problems within the system are provided. Best operating conditions and limitations of the system are deduced and explained using these results.

Chapter 6 entitled "Conclusion" gives a summary of each previous chapter and of the results achieved, followed by suggestions of possible future work that may make the system free from its apparent limitations. It then goes on to conclude this report.

2 LITERATURE REVIEW

This chapter summarises all the background reading done to gain enough insight into computer vision to implement this project. It starts off with an introduction to computer vision and a brief history of it. It then goes on to explain the fundamental principles that are the basis of this project – multi-view relations.

It then introduces the concept of epipolar geometry which is a relationship between two images of the same scene. Equations describing the epipolar relation are explained and methods of computing the epipolar geometry of a scene are discussed. The RANSAC algorithm is explained at the same time because of its application in the computation of epipolar geometry of scenes as well as other computations required in this project.

Planar homography, another relationship between images of a scene is explained next. Its applications and method of computation (which uses RANSAC again) is also discussed in great detail. Both epipolar geometry and planar homography are essential in finding point correspondences between the images on a scene which is vital for obstacle detection.

For the computation of the epipolar geometry and planar homography of a scene, a set of initial point correspondences are required between images. Corners are usually the most suitable points to use as candidate correspondences for such computation. Therefore corner detection is studied as well in the literature review. Once the epipolar geometry of a scene has been computed, further correspondences which are necessary for obstacle detection can be found using the solution. Points on contours of objects are the most reliable for this purpose (bearing in mind corner correspondences have already been found). Therefore edge detection techniques are briefly discussed as well before selecting the most appropriate one for the purposes of this project.

Finally, several processes of image segmentation are discussed before the selection of the most appropriate one for this project. Planar homography and epipolar geometry find point correspondences between images of a scene and can be used to roughly estimate the height of points on the scene. However, image segmentation is necessary to separate objects on the scene (whether they are on the ground or obstacles) and then multi-view relations can be used to find the heights of objects as a whole before identifying them as possible obstacles.

2.1 Computer Vision and a brief history

Computer vision is a subfield of artificial intelligence in computer science. It is often described as essentially the emulation of human vision by a machine. This is not considered as entirely accurate by many computer scientists since despite both forms of vision being reliant on light radiated from the environment, the principles behind the implementation of human and computer vision are far from similar.

In order to perceive the world through vision, the machine must be able to recognise certain characteristics of its immediate environment. This is done through the analysis of images taken by a camera. The ability to recognise and differentiate textures, shapes, colours etc is vital to the replication of human vision which is the ultimate goal of all computer vision research. Perhaps the greatest challenge to researchers in the field is the interpretation of 2D images into 3D scenes since a significant proportion of a scene's characteristics are absent in its image. Moreover, a prior knowledge of the environment (e.g. lighting, texture, background etc.) is required since they affect the appearance of objects which the computer is trying to describe [16].

Computer vision problems can be subdivided into low-level ones and high level ones [16]. The former refers to the initial stages of image processing with very little other knowledge of the environment and includes operations such as edge detection, boundary drawing etc whereas the latter refers to highly task-specific operations performed on presumably perfect data from low-level processes with some or no additional data and includes operations such as obstacle detection, face recognition etc.

A Brief History

Researchers have taken a bottom up approach in computer vision for decades starting with algorithms that performed very simple analyses of raw input images direct from cameras [16]. Roberts [19] is well known for his work in the blocks world, which is basically the concept that aritifical environments only consist of regular geometric shapes. His work mainly consisted of algorithms that converted the input image into a large number of line drawings and then analysed sets of lines to identify and differentiate among pre-defined geometric shapes.

Image segmentation, i.e. the division of an input image into several distinguishable segments with common characteristics is another low-level process vital to computer vision. Brice and Fennema [3] in 1970 developed one of the earliest algorithms to accomplish these objectives. The algorithm involved identifying regions that contained similar pixels and then merging together similar regions.

Obviously one of the problems with image segmentation is the possibility of under-segmentation (i.e. boundaries are not recognised and different regions can be identified as one) or over-segmentation (i.e. a single region has been identified as several segments instead of one because of a false boundary) in the absence of sufficient information about the scene. Among other things, shadows and varying illumination often contribute to these inconsistencies. To account for such inconsistencies Guzman in 1968 [10] introduced heuristics that could segment 2D images into meaningful 3D interpretations. Tenenbaum and Barrow [23] on the other hand introduced the application of object-to-object relations to segment objects in a scene in 1976.

The decade after that most contributions to computer vision came from the application of physics and geometry [16]. Information on shapes within the scene such as convexity, concavity, slopes and smoothness would in turn provide information on the scene itself. In 1970 Horn [13] incorporated the analysis of texture and shading to help represent the scene into a 3D form within the machine. Once again Barrow and Tenenbaum [1] worked with reflectances and orientation information from the input image towards the same objective in 1978.

Scene analysis i.e. the reconstruction of a representation of the image from a simple description of the scene (usually by identifying geometric properties of objects present). Other than geometric features, texture, markings and colour also contributed to the representation of the scene with the computer which was then used in a variety of applications such as obstacle detection and avoidance, navigation, planning etc.

2.2 Multi-view relations

Multi-view relations are the fundamental principles on which this project is based. These include epipolar geometry and planar homography. The fact that different views of the same are somehow related is the basis behind multi-view relations. In fact several relationships exist between images taken of the same scene, which have been researched for decades and these relationships have ultimately contributed significantly towards computer vision. A range of tasks and algorithms have been developed over the years that have aided computer scientists in their research of computer vision. These algorithms give a modern computer vision system the following abilities:

- Given merely a pair of images, the ability to find matches between 3D positions of points between the two images
- Given merely three images, the ability to compute matches between 3D positions of points and lines between the images
- The ability to compute the epipolar geometry of the scene without the need of calibrating through an object
- The ability to compute the internal calibration of the camera from a series of images.

from Hartley [12]

What makes these algorithms unique is the fact they are uncalibrated, that is the internal parameters of the camera are not required at all for these computations.

Two-view geometry

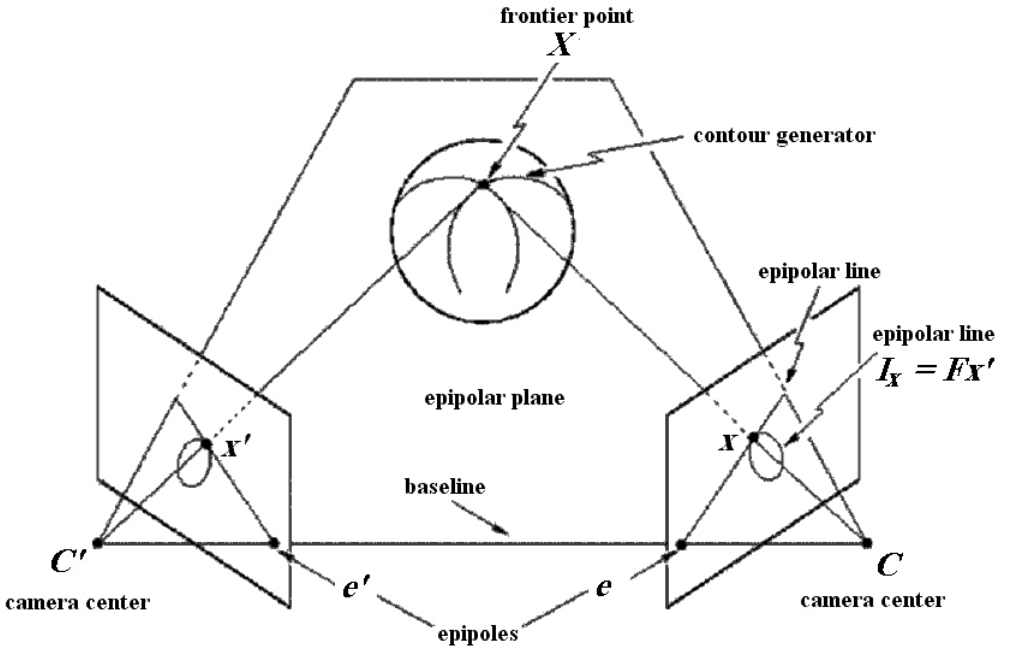
The relationship between two images of the same scene can be more than enough to accomplish the task of obstacle detection to a reasonably reliably extent. The pair of images can be obtained at the same time using a stereo rig (i.e. using two cameras) or one after the other, usually with a single camera moving relative to the scene.

2.3 Epipolar Geometry

Epipolar geometry is defined "the intrinsic projective geometry between two views" by Hartley [12]. The reason epipolar geometry is being studied here is because of its applications in finding point correspondences between images (i.e. 2D co-ordinates of the same point on a scene on images taken from two different camera positions). Epipolar lines (Section 2.3.1) are vital in narrowing down the possible candidate matches on one image for a point in another image of the same scene. Planar homography (Section 2.4) can be used to estimate the heights of these points (usually on edges) performing obstacle detection.

The epipolar geometry of a scene is not at all dependent on its structure but rather on the camera's internal parameters and relative pose (although these do not have to be known beforehand). The epipolar geometry can be expressed as its fundamental matrix. This matrix, denoted as F, is a 3x3 rank 2 matrix. The theory of epipolar geometry states that given a set of matching points x and x' (in 3 space) obtained from two separate views of the same scene, the points satisfy the relation

$$\boldsymbol{x'^{\mathrm{T}} F x = 0}$$



Adapted from Mendonça [17]

Figure 2.1: Epipolar geometry of a scene from two images
X is the location of a point on the scene and x' and x are its locations
on the images planes from camera centres C' and C respectively

The **baseline** can be defined as the line joining the two camera centers.

The **epipoles** can be defined as the points of intersection of each image plane (for each view) and the baseline.

The **epipolar plane** can be defined as a plane containing the baseline. There may be a series of such planes depending on a single parameter

An **epipolar line** is basically the intersection between the image plane and one of these epipolar planes, also intersecting at the epipole. Kasturi's definition: "An epipolar line intersects the left and right image planes in epipolar lines and defines the correspondence between them" [14].

2.3.1 The fundamental matrix and epipolar lines

As mentioned earlier the fundamental matrix satisfies the following condition for any pair of corresponding points between two images of the same scene

$$x'^{T}Fx = 0$$

Some properties of the fundamental matrix [12]

- The transpose of F represent the relationship between the two points x' and x, in the opposite order. i.e. $x^{T}F^{T}x' = 0$
- $I = F^{T}x'$ represents the epipolar line corresponding to the point x in the second image. That is the corresponding point x on the second image satisfies $x^{T}I=0$.

- The line $I'=Fx$ contains the epipole e' for any point x on the second image (other than e). Similarly the line $I=F^Tx'$ contains the epipole e. i.e. $e'^T(Fx)$ for all values of x and thus $e'^TF=0$ and similarly $Fe=0$.
- With nine elements (being a 3X3 matrix) and the common scaling not being important F would have eight degrees of freedom. However owing to the constraint that det(F)=0, F has seven degrees of freedom
- F is not of full rank since inverse mapping is not possible. This is because of the fact that with the pair of epipolar I' and I lines corresponding to matching points, any point on the second image on I is mapped on the same line I'.

From the relationship
$X'^TFX = 0$ where
$X = [x\ y\ 1]^T$ and $X' = [x'\ y'\ 1]^T$ and

$$F = \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{33} & f_{32} & f_{33} \end{pmatrix}$$

We have

$$[f_{11}x'+f_{21}y'+f_{31}\ \ f_{12}x'+f_{22}y'+f_{32}\ \ f_{13}x'+f_{23}y'+f_{33}]\ [x\ y\ 1]^T = 0.$$

So $$(f_{11}x' + f_{21}y' + f_{31})x + (f_{12}x' + f_{22}y' + f_{32})y + (f_{13}x' + f_{23}y' + f_{33}) = 0$$

Therefore given the co-ordinates of a point X' in the first image and the fundamental matrix for the scene, it is possible to compute a line (**the epipolar line**) in the second image on which the corresponding point X lies. This is basis used for finding matching points between two images along their contours used in this project and it will be essential in detecting the heights of these points above the ground plane which is effectively performing the task of obstacle detection.

2.3.2 Computing the Fundamental Matrix

The following shows how an ordinary least squares solution can be found to the computation of the fundamental matrix.

We have the epipolar relation
$\boldsymbol{X'^TFX = 0}$
where $X = [x\ y\ 1]^T$ and $X' = [x'\ y'\ 1]^T$

$$F = \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{33} & f_{32} & f_{33} \end{pmatrix}$$

We have $X'^TFX = 0$

$$[f_{11}x'+f_{21}y'+f_{31}\ \ f_{12}x'+f_{22}y'+f_{32}\ \ f_{13}x'+f_{23}y'+f_{33}]\ [x\ y\ 1]^T = 0$$
$$f_{11}x'x + f_{12}x'x + f_{13}x' + f_{21}y'x + f_{22}y'y + f_{23}y' + f_{31} + f_{32}y + f_{33} = 0$$
$$[x'x\ \ x'y\ \ x'\ \ y'x\ \ y'y\ \ y'\ \ x\ \ y\ \ 1]f = 0$$

$$Af = \begin{pmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{pmatrix} f = 0$$

2.3.2.1 RANSAC

The RANSAC (Random Sampling Consensus) algorithm is "an algorithm for robust fitting of models in the presence of many data outliers" [9]. The RANSAC algorithm is studied here because of its applications in the computations of parameters for several models required for obstacle detection (epipolar geometry and planar homography).

Any sample of data may have a number of outliers that do not correspond to the intended model. The classification of data into inliers and outliers can be a problem. Given a set of n data points, if the number of outliers is known as x, then the parameters of the model can be estimated repeatedly with combinations of (n-x) elements and keeping track of how many points agree with each of these models. Points agreeing with a model are those that are within a threshold of the model (calculated using a mathematical function e.g. Euclidian distance). These points are called the consensus set. However this method is not always achievable when the number of outliers is not known. The RANSAC algorithm does not have this drawback.

The method involves selecting a certain minimum number of elements from the data sample randomly. Then the parameters of the model are estimated using these elements. Then the support for the estimated model (i.e. the number of elements in the consensus set) is noted by computing how many points are within the threshold. This process of random selection, parameter estimation and support computation is repeated a number of times and then the model with the most support is chosen. The parameters of the model are now estimated again but this time with all the elements in the consensus set of the previously chosen model. This final model is the output of the RANSAC algorithm.

Algorithm

i. Randomly select a sample of s data points from S and instantiate the model from this subset
ii. Determine the set of data points S_i, which are within a distance threshold t of the model. The set S_i is the concensus set of the sample and defines the inliers of S
iii. If the size of S_i is greater than some threshold T, re-estimate the model using all the points in S_i and terminate
iv. If the size of S_i is less than T, select a new subset and repeat the above.
v. After N trials the largest consensus set S_i is selected and the model is re-estimated using all the points in the subset S_i

Adapted from Hartley [12]

Figure 2.2: Algorithm for RANSAC

2.3.2.2 Normalised 8 point algorithm

This is the simplest method of computing the fundamental matrix. It involves first normalising[1] the points, computing the value of F using a linear equation and enforcing the singularity constraint on it and then denormalising F to give the actual fundamental matrix.

The normalised 8-point algorithm is obviously not optimal since the significance of all entries of F is not equal i.e. some entries are more constrained by the point-point matches than others [12]. Several optimisations can be performed to computer a better estimate of F, such as iterative estimation which is based on minimising the algebraic error and the Gold Standard method which is dependent on the assumption of an error model.

However, the simplest and easiest method for the automatic computation of F is the RANSAC method using eight point-correspondences. Please note that only seven points are required to compute F and eight points would increase the order of the computation of F exponentially. However, using seven points we might end up with three solutions for each F which is not desirable when using a large number of combinations of point-correspondences to repeatedly compute F.

A random sample of eight correspondences is first chosen and the fundamental matrix is computed using them. The distance for each correspondence is then calculated and the number of inliers is deduced (i.e. number of correspondences for which distance is within the threshold). This process is repeated a large number of times and the F with the largest number of inliers is chosen for the following operations. F is now estimated using all correspondences that were classified as inliers for the previously chosen F.

Algorithm

i. Interest Points: Compute interest points in each image
ii. Putative Correspondences: Compute a set of interest point matches based on proximity and similarity of their intensity neighbourhood.
iii. RANSAC robust estimation: Repeat for N samples where N is determined adaptively
    a) Select a random sample of 8 correspondences and compute the fundamental matrix F.
    b) Calculate the distance d for each putative correspondence
    c) Compute the number of inliers consistent with F by the number of correspondences for which d < t pixels

    Choose the F with the largest number of inliers. In the case of ties choose the solution that has the lowest standard deviation of inliers.
iv. Non-linear estimation: re-estimate F from all correspondences classified as inliers by minimizing a cost function
v. Guided matching: Further interest point correspondences are now determined using the estimated F to define a search strip about the epipolar line

Adapted from Hartley [12]

Figure 2.3: Algorithm for the normalised 8-point method of computation of F

Normalisation[1] – This is the translation and scaling of the image such that the centroid (average position) of the reference points is at the origin and the RMS distance of the points from the origin is equal to √2 [12]. This has the effect of removing the effects of the camera calibration.

2.4 Planar Homography

Planar homography can be described as the relationship between corresponding points between two images of the same scene. Planar homography can be used to identify points on the same plane of an image. In this project, it is essential in identifying the ground plane and to find the heights of distinct points above it (Section 2.4.2). The computation of the homography of a scene is also necessary to detect obstacles via image warping (Section 2.4.3) which is one of the three methods used in this project.

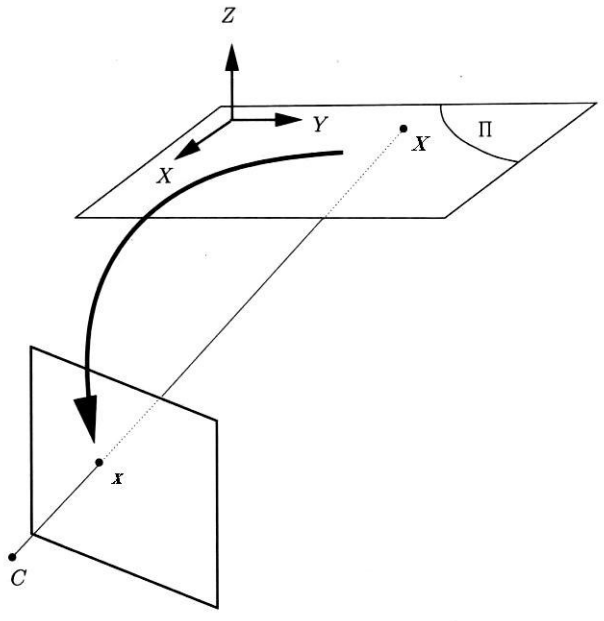
The homography relationship is encapsulated by the *H* matrix such that for points *X'* and *X* on the two images the following holds:

$$X \approx HX'$$

Given the matrix *H* for a certain plane on the scene, the relationship holds for all points on the image that are on the same plane.

Homography by itself is the relationship between the plane in space and the image plane (Figure 2.4). Let *X* be the co-ordinates of a point on the image plane while *X'* are its co-ordinates on the plane in space II.

$$\underset{X}{[x\ y\ z]^T} \approx P\underset{X'}{[x'\ y'\ 0\ T]^T} T = P[X\ Y\ T]^T$$



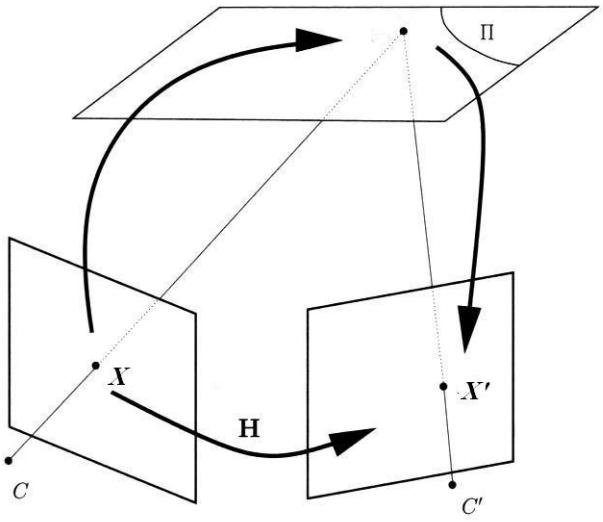
Adapted from Faugeras [8]

Figure 2.4: Homography relationship between a plane in scene and image plane

2.4.1 Homography between two images of a plane

Given two images of the same scene, it is already known that there exists a a relationship or homography between the image plane of the first camera (position) and the plane in space and that between the image plane of the second camera (position) and the plane in space. It can thus be deduced that there is a homography between the two image planes. This relationship is known as planar homography. Thus there exists the following relationship between a pair of corresponding points (*X'* and *X* on different images) on the same plane used to compute the homography.

$$X' = HX$$



Adapted from Faugeras [8]

Figure 2.5: Homography relationship between two images of scene

Thus if H is known for a particular plane between two images, for any point X on the first image, the position of its corresponding point X' on the second image can be computed given that X lies on the same plane the homography is related to. Similarly the converse can be done using

$$\boldsymbol{X = H^{-1}X'}$$

2.4.2 Homography between parallel planes of a scene

Given the homography matrices Hz_1 and Hz_2 for two parallel planes that have distances z_1 and z_2 from the camera respectively, the homography matrix for a third plane z parallel to the given planes can be computed using interpolation [24]:

$$H_z = \frac{\frac{1}{z} - \frac{1}{z_2}}{\frac{1}{z_1} - \frac{1}{z_2}} \cdot H_{z_1} + \frac{\frac{1}{z_1} - \frac{1}{z}}{\frac{1}{z_1} - \frac{1}{z_2}} \cdot H_{z_2}$$

This means that given the homographies of two parallel planes, the homographies of a number of planes (parallel to the given pair) at different heights can be computed. Given the locations of a point between two images of the scene, it is possible to identify the homography the point corresponds to via comparison to its "warped" position using the homography relation ($X' = HX$). Since we know the height of each plane (which was used to compute each homography), the height of the point can simply be estimated as the height for the homography it most closely corresponds to.

2.4.3 Image Warping

When two images are taken of a scene from different camera positions, the relationship $\boldsymbol{X' = HX}$ can be used to create a "warped" image from the first image. The warped image is essentially an "estimate" of the second image given that that all points on both images lie on the same plane as the ones used to compute the homography of the scene.

While creating the warped image, the warped coordinates X of each pixel with coordinates X' on the first image is found using $\boldsymbol{X' = HX}$. The intensity of X (or intensities if using RGB images) are copied to position X' on the warped image. However this means that there may be pixels on the warped image which are not warped positions for any pixels in the first image and there may be pixels that are warped positions for more than one pixel from the first image. These problems are solved using interpolation. Blanks pixels are simply filled up by averaging the intensities of their non-blank neighbours. Pixels that are warped positions for more than one pixel on the first image have the average intensities for all the corresponding pixels from the first image.

Assuming the plane to which the homography corresponds to is the ground, the warped image and second image should be identical except for parts of the scene that are above the ground plane (i.e. obstacles). The difference between intensities of corresponding pixels between the warped image and second image is often used as a method of obstacle detection.

2.4.4 Computing the homography matrix

By using the scale ambiguity of the homography equation (e.g. $h_9 = 1$) we can turn the computation of H into a OLS (ordinary least squares) problem [21]. This is the **inhomogeneous solution** to the computation of H.

From $\boldsymbol{X' = HX}$
where $X = (s\ t\ 1)'$ and $X' = (x\ y\ 1)'$

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{pmatrix}$$

We have

$$\begin{pmatrix} s \\ t \\ 1 \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} s \\ t \\ 1 \end{pmatrix} = \begin{pmatrix} xh_1 + yh_2 + h_3 \\ xh_4 + yh_5 + h_6 \\ xh_7 + yh_8 + 1 \end{pmatrix}$$

$$s = xh_1 + yh_2 + h_3 \quad \text{i)}$$
$$t = xh_4 + yh_5 + h_6 \quad \text{ii)}$$
$$1 = xh_7 + yh_8 + 1 \quad \text{iii)}$$

Multiplying i and iii we have
$s(xh_7 + yh_8 + 1) = (xh_1 + yh_2 + h_3)1$
$sxh_7 + syh_8 + s = xh_1 + yh_2 + h_3$
$-xh_1 + -yh_2 + -1h_3 + 0h_4 + 0h_5 + 0h_6 + sxh_7 + syh_8 = -s$

Multiplying ii and iii we have
$t(xh_7 + yh_8 + 1) = (xh_4 + yh_5 + h_6)1$
$txh_7 + tyh_8 + t = xh_4 + yh_5 + h_6$
$0h_1 + 0h_2 + 0h_3 + -xh_4 + -yh_5 + -1h_6 + txh_7 + tyh_8 = -t$

The two equations per point would now read.

$$\begin{pmatrix} -x & -y & -1 & 0 & 0 & 0 & s*x & s*y \\ 0 & 0 & 0 & -x & -y & -1 & t*x & t*y \end{pmatrix} \begin{pmatrix} h_1 \\ .. \\ .. \\ h_8 \end{pmatrix} = \begin{pmatrix} -s \\ -t \end{pmatrix}$$

So for *n* points it may be written as

$$\begin{pmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & s_1*x_1 & s_1*y_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & t_1*x_1 & t_1*y_1 \\ : & : & : & : & : & : & : & : \\ : & : & : & : & : & : & : & : \\ -x_n & -y_n & -1 & 0 & 0 & 0 & s_n*x_n & s_n*y_n \\ 0 & 0 & 0 & -x_n & -y_n & -1 & t_n*x_n & t_n*y_n \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{pmatrix} = \begin{pmatrix} -s_1 \\ -t_1 \\ : \\ : \\ -s_n \\ -t_n \end{pmatrix}$$

tmp1 *htmp* *tmp2*

$tmp1 * htmp = tmp2$
$htmp = tmp1^{-1} * tmp2$

$$H = \begin{pmatrix} htmp_1 & htmp_2 & htmp_3 \\ htmp_4 & htmp_5 & htmp_6 \\ htmp_7 & htmp_8 & 1 \end{pmatrix}$$

Please note that a satisfactory solution can not be obtained using this method if $h_9 = 0$, however this is the method most commonly used because of the rarity of such circumstances. Since there are eight unknowns, four point correspondences are required to solve the computation of H. However if any three of these points are collinear the solution would not be satisfactory. This is because the effect of having a set of collinear points in solving linear equations is the same as having a pair of points since either can define a straight line and neither can define a plane.

As with the computation of the fundamental matrix, a RANSAC method is perhaps the most suited one for the computation of the homography of a scene.

Algorithm

i. Interest Points: Compute interest points in each image
ii. Putative Correspondences: Compute a set of interest point matches based on proximity and similarity of their intensity neighbourhood.
iii. RANSAC robust estimation: Repeat for N samples where N is determined adaptively
   a) Select a random sample of 4 correspondences and compute the fundamental matrix H.
   b) Calculate the distance d for each putative correspondence
   c) Compute the number of inliers consistent with H by the number of correspondences for which d < t pixels

   Choose the H with the largest number of inliers. In the case of ties choose the solution that has the lowest standard deviation of inliers.
iv. Non-linear estimation: re-estimate H from all correspondences classified as inliers by minimizing a cost function

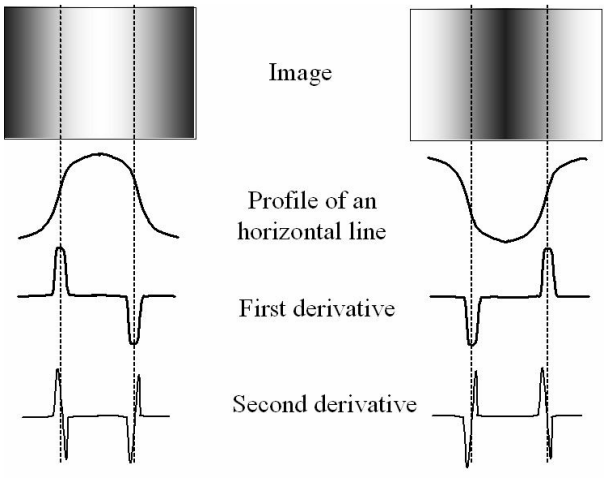
Adapted from Hartley [12]

Figure 2.6: Algorithm for computation of H using RANSAC

2.5 Edge Detection

Edges are characterised by abrupt changes on an image indicating boundaries or other features. The detection of edges is vital in computer vision in accomplishing several operations such as segmentation, object modelling, registration etc. Points on edges are used in this project to find point correspondences along contours of obstacles using epipolar geometry (Section 2.2). Homography (Section 2.3) can be used on these correspondences to estimate the heights of these points on the scene thereby identifying obstacles.

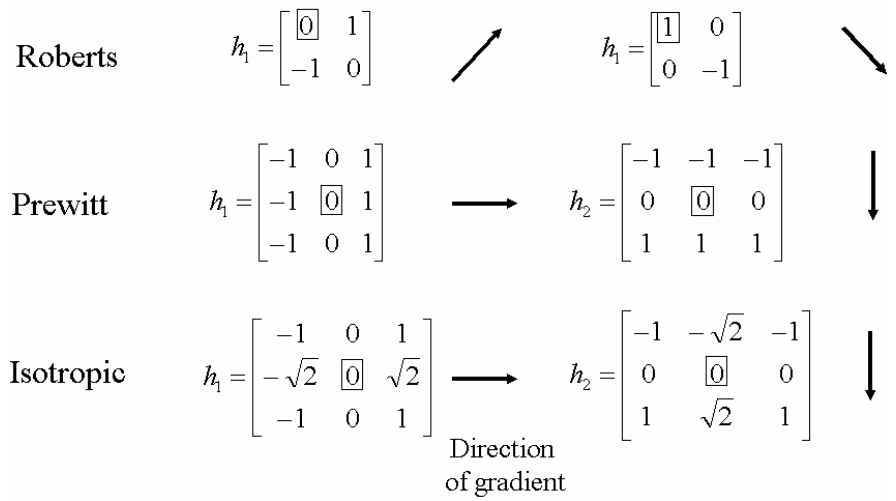
Figure 2.7 shows parts of two images with varying intensity along the horizontal. The intensity function of each pixel and its derivatives are vital in detecting edges on any object as is obvious by observing the distinct characteristics of these derivatives at edges [6].



from CVI [6]

Figure 2.7: Edges and derivatives of their intensity

Several edge detection methods are used to detect edges. The Sobel, Prewitt and Roberts [19] methods find edges using their approximations to the derivative returning points where the gradient is maximum as edges [6]. The common gradient operators used by each of these methods along with the direction of edges are shown in Figure 2.7



from CVI [6]

Figure 2.8: Edge detection methods and their common gradient operators

The Laplacian of Gaussian method is based on locating points by looking for zero crossings once the image has been filtered by a Laplacian of Gaussian filter. It looks at the second order derivatives of points in order to identify zero-crossings [2].

However all these methods are very susceptible to noise. Several performance criteria for good edge detection were introduced by Canny [4]. These included good detection (i.e. minimum points really on edges overlooked), good localisation (the coincidence of identified edge points to the real edge) and unique response (the same single set of points were recognised as edges of an image regardless of the number of runs).

The Canny method [4] thus satisfied these criteria as much as possible. Using this method the gradient is first calculated using the derivative of a Gaussian filter. Local maxima of the gradient of the image are then identified as edges. The method also uses a pair of thresholds to identify strong and weak edges. Strong edges and only those weak edges that are connected to strong ones are indicated as edges in the final output. Because of its ability to distinguish weak edges from strong ones this method is the least susceptible to noise [2].

2.6 Corner Detection

Corners are local features on images identified by large variations in intensity in both x and y directions. Corners detected during image analysis are used in a variety of applications such as stereo point correspondence location, motion tracking, image database retrieval etc since they are most reliable features on image in terms of the lack of ambiguity. This means that motion is very distinct at corners and moreover corners can also be used in the reconstruction of most objects even though it would be a mere approximation. Point correspondences are required for the computation of the homography (Section 2.4.4) and epipolar geometry (Section 2.3.2) of a scene. Corners are usually selected as the points used since they are the most distinguishable points and thus the easiest to match across images.

Corners are detected using derivatives of the intensity function of pixels in both x and y directions. The most commonly used corner detection algorithms (KLT in US and Harris [11] in Europe) compute the "local structure matrix" of each pixel to determine whether it is a corner.

For a pixel (x,y) with intensity function $I(x,y)$, its local structure matrix (A) is

$$A = \begin{bmatrix} \left(\frac{\partial I(x,y)}{\partial x}\right)^2 & \left(\frac{\partial I(x,y)}{\partial x}\right)\left(\frac{\partial I(x,y)}{\partial y}\right) \\ \left(\frac{\partial I(x,y)}{\partial x}\right)\left(\frac{\partial I(x,y)}{\partial y}\right) & \left(\frac{\partial I(x,y)}{\partial y}\right)^2 \end{bmatrix}$$

In the presence of noise the image is often smoothed before computing the local structure matrix for each pixel. Furthermore each of the entries may be smoothed by a Gaussian filter $w(r;s)$ with a preset s or a simple box filter.

Note that A is always symmetric and positive semi-definite and it has exactly two eigenvalues λ_1 and λ_2 (i.e. $\lambda_1>0$ and $\lambda_2>0$) [25]

These eigenvalues can then be used to distinguish between uniform points, edges and corners on the image.

- For points that are on perfectly uniform parts of the image $\lambda_1 = \lambda_2 = 0$
- For points that are on perfect black and white step edges $\lambda_1 > 0$ and $\lambda_2 = 0$ where the eigenvector (v_1) corresponding to λ_1 is orthogonal to the edge.
- For points that are on perfect black square against white background corner, $\lambda_1 > 0$ and $\lambda_2 > 0$

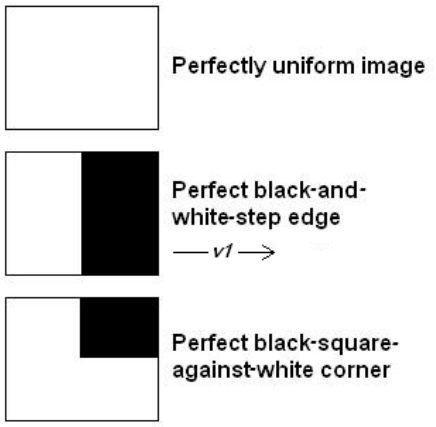


Figure 2.9: Corners, Edges and other points

This is the basic principle the Harris corner detector [11] uses while working on greyscale images. The algorithm for this method is as follows

Algorithm
- For each pixel *(x,y)* find A
- For each A find 2 eigenvalues λ_{max}, λ_{min}
- Sort all λ_{min}, discard pixel with small λ_{min}.
- Discard pixels with large $\lambda_{max} - \lambda_{min}$
- Remaining are corner points

From Wong [25]

Figure 2.10: Algorithm for the Harris Corner Detector

2.7 Image Segmentation

Image segmentation may be defined as the process of distinguishing objects in an image from the background. The purposes of segmentation in obstacle detection are quire obvious, since segments must be distinguished or separated first before they can be identified as obstacles.

Typically several techniques are used to accomplish this task when dealing with images that are described by the intensity of each pixel (intensity images) [22].

- **Edge-based methods**: These methods are completely dependent on the system's ability to detect contours of objects. Pixels that are enclosed by the same contours lines are then identified as a single object. This method is not reliable at all in the presence of blurring because the contour lines formed are incomplete. As a result pixels from different actual objects may be grouped together through the gap in the contours.
- **Threshold techniques**: These methods rely on the ability to distinguish objects from the background only if the intensity of the pixels on it fall outside a certain threshold which is a characteristic of the background. This means that pixels with intensities that are outside the range of intensities of those on the background are treated as objects on it. This method too is very susceptible to blurring since only local rather than spatial information on pixels is taken into account. Moreover a prior knowledge of the background itself is required to preset the allowable range of intensities on it, unless some other methods are used to first identify the ground itself or to first train the system on each particular background.

- **Region based method**: This method involves first segmenting the image into connected regions by grouping neighbouring pixels that have similar intensities [3]. Regions next to each other can then be merged together depending on how similar they are or how sharp the boundaries are between then. Obviously the stringency of these criteria for merging regions can lead to inaccurate segmentation due to under-merging or over-merging [22].

- **Connectivity-preserving relaxation method**: This recently proposed method based on the identification of initial curve boundaries and their iterative modification through the application of various shrinking and expanding operations dependent on a certain energy function [7]. As is expected of such methods the system would still be very liable to getting caught in a local minimum.

The region based method is probably the simplest method capable of decently identifying objects on a background. Furthermore, it is less susceptible to noise and broken edges and no prior knowledge of the scene is required as in threshold techniques. On the other hand, the connectivity preserving relaxation method is still a relatively new method and was considered too complex for the purposed of this project.

2.7.1 K-means clustering

Region based image segmentation would probably be the most widely used technique given minimal information about the scene, i.e. with no prior knowledge of the range of intensities of pixels on the background. However first pixels of similar intensities are needed to be grouped together. It is understandable that even on an object with little or no texture, pixels will not have the same intensity. Therefore some classification method is necessary to first group together pixels that are similar in intensity making the following process of segmentation less likely to err.

K-means clustering is perhaps the most suitable method for this task especially since it is an unsupervised method. The basic idea behind this method is being given a set of pattern vectors $T = \{x_1,x_2.....x_n\}$, each vector is assigned a cluster c_i from a set of k possible classes where $c_i=j$ implies vector x_i belongs to cluster j. It starts off by initialising class centres (m_c's) and then assigning each pattern vector to its closest class in terms of distance or some other measurement. The class centres are then updated averaging the values of the vectors within it and vectors are then reassigned to their closest classes using recomputed class centres. The process is iterative and only terminates once there are no changes in assignments of all pixels following updates of class centres.

1. Initialise class centres $\mathbf{m}_1, \ldots, \mathbf{m}_k$.
2. For each datapoint $\mathbf{x}_i$, assign it to the same cluster as the *closest* cluster centre (in the sense of minimum squared distance), i.e. $c_i = \arg\min_j d(\mathbf{x}_i, \mathbf{m}_j)^2$.
3. Recompute the cluster centres as the mean of all the points that belong to the cluster; $\mathbf{m}_j = \sum_{i.c_i=j} \mathbf{x}_i / n_j$.
4. Repeat at 2 until the cluster assignment does not change.

from PAT [18]

Figure 2.11: Algorithm for K-means clustering

The k-means algorithm is not without its disadvantages. Class centres need to be chosen during initialisation (usually as random members of the dataset). This means the final result is dependent on the selection of these initial class centres. Moreover during the classification process it is possible that some clusters may become empty eventually. This would once again give unsatisfactory results. The

algorithm converges to a local minimum rather than a global minimum since there is the possibility of a better assignment of classes with different class initialisations that lead to a better solution. Prior knowledge on the distribution of the dataset is often used to gain better results in terms of a global minimum.

For the purposes of region based segmentation, class centres are initialised with intensity values rather than co-ordinates. A pixel is assigned to a class with central intensity closest to its own relative to that of other class centres. The class centres are then recomputed averaging the intensities of all pixels within the cluster.

Following the process of k-means clustering region-based segmentation becomes a considerably simpler problem to tackle. Following classification via k-means, adjacent pixels assigned to the same cluster are identified as part of the same segment. The process can be reiterated several times until no neighbouring pixels with the same class label are parts of different segments.

2.8 Summary of review

This chapter has gone over all the concepts vital to this project in detail. It has covered all the background knowledge in the fields of computer vision, pattern recognition and image processing that is necessary for the design and implementation of the intended system.

Multi-view relations (planar homography and epipolar geometry) have been explained along with their methods of computation. Processes such as corner detection that are necessary for preparing the inputs for these computations have been discussed. Other processes such as image segmentation and edge detection that are vital to obstacle detection have also been gone though.

The next chapter entitled “Design” identifies the precise requirements of this intended system and gives a brief outline of the plan of its implementation using the information on concepts and techniques covered in this chapter.

3 DESIGN

This chapter first attempts to come up with a more comprehensive list of requirements for the system intended by this project. There is also a small discussion on the several design methodologies commonly used to develop system and a choice is made after considering the timeframe and nature of the project. Other factors affecting the design of the project such as hardware constraints are mentioned along with a discussion on the development tools and methods used to implement it The problem is finally broken down into a series of smaller and smaller sub-problems. An analysis of these sub-problems as well as their interdependencies as part of the main problem are then discussed as part of the "Design Plan".

3.1 Requirements

As was mentioned earlier in the introduction, the main aim of this project is to use several techniques to research and study the reliability with which a computer can detect obstacle given certain constraints in hardware. It uses a combination of very fundamental techniques in various subfields of computing that are later amalgamated into the obstacle detection machine that is the ultimate product of this project. Needless to say a bottom-up approach has been taken to develop this system. It may be noted that this project is not simply an engineering or development project and it may be categorised more as research since most of it emphasises on the comparisons between several computer vision techniques in terms of reliability.

The final system would use image processing techniques such as detecting corners and edges, pattern recognition techniques such as clustering and segmentation and finally the basic concepts of computer vision based around multi-view relations (planar homography and fundamental matrices) to study how capable a machine is in accomplishing the seemingly simply task of obstacle detection.

The following is a very basic list of requirements that the aimed system was intended to satisfy

i. To process images and find correspondences of corners between them
ii. To distinguish the ground plane from all other planes given two images of the same scene
iii. To identify points on either image belonging to the same plane
iv. To form and compare a "warped" image which can be described as an estimation of the final image (made using the initial image) assuming everything on the scene is on the ground.
v. To calculate the height of points on either image given that the ground plane has been identified
vi. To segment individual objects on the scene such that the height of each object can be found
vii. To process images and find edges and their correspondences between them
viii. To find the height of points along the edges of an obstacle such that it is useful for detecting slopes or spherical objects

The true purpose of this system is the evaluation of the three obstacle detection techniques mentioned earlier in the introduction, which have overlapping requirements. Here they are again along with their specific requirements from the basic list given above:

- Obstacle detection using planar homography and image warping (via comparison to warped image) – i, ii, iii, iv
- Obstacle detection using planar homography and image segmentation (via computation of heights of segments) - i, ii, iii, v, vi
- Obstacle detection using epipolar geometry, planar homography and edge detection (via computation of heights along contours) - i, ii, iii, vii, viii

Please note that this project does not aim to accomplish these tasks in real time since its main purpose is simply the evaluation of the reliability of these techniques. Since the fundamental concepts behind it

are multi-view relations, pictures are simply taken using a digital camera from two different positions (assuming pure translation between them) and then the set of consecutive images are transferred on to a standard personal computer which processes these images attempting to accomplish the tasks specified above.

3.2 Design Methodology

A bottom-up approach was taken to develop the system intended. A bottom-up approach implies that the main problem (i.e. obstacle detection) was broken down into smaller and smaller subproblems. The most basic subproblems were tackled individually and their solutions were put together into progressively more complex subproblems which were finally integrated to form the final system.

The top-down approach of implementing the basics of the main problem first and then identifying and solving smaller sub-problems progressively seemed unsuitable for the purposes of this project. The basis behind this reasoning is that without the solution to the most basic sub-problems (e.g. detecting corners and finding point-correspondences) upon which all other sub-problems are dependent it would be rather pointless defining the main problem. It would have possible to do so, if sample solutions (e.g. matched corners) were used as test data for the larger sub-problems, however enough sample solutions were simply not available to test completely the functionality and accuracy of the system. Therefore, the basic sub-problems had to be solved first and then they were used to solve and test progressively larger sub-problems.

System Design Model

Even though there are a number of system design models only two were considered for this project. These were the waterfall model and the spiral model.

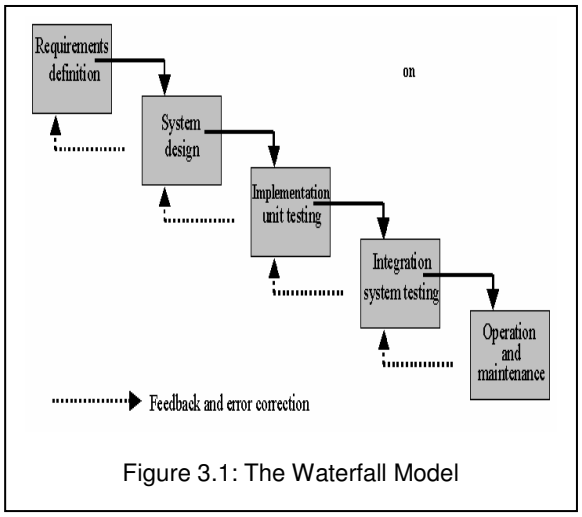


Figure 3.1: The Waterfall Model

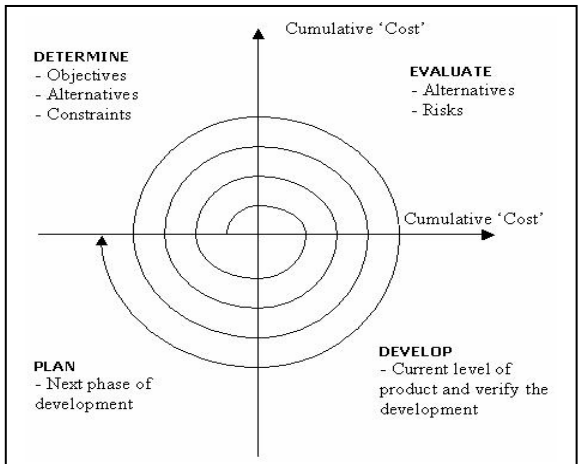


Figure 3.2: The Spiral Model

The waterfall model is a very generic step by step model with feedback and error loops going back to the previous step in case changes need to be made. Although it was the most commonly used model earlier, it should only be treated as a guideline since it is not a very realistic model. It is perhaps only applicable for short projects that do not necessitate or can not accommodate much feedback or review in between the steps.

The spiral model on the other hand is more suited to longer term projects. It is a process of iterative development where the parts of the problem are continuous analysed, evaluated and developed with continuous review. Given the bottom-up nature of this project, the spiral model was considered appropriate, since it allows the individual design and implementation of sub-problems which is an

incremental process. The research nature of this project means that new features are continuously being added to it which makes iterative development a necessity.

3.4 Hardware Constraints

Before the design plan can be discussed, the constraints on the system must be considered. As it was mentioned a very standard personal computer is used to process images transferred from a digital camera rather than attempting to process them via a webcam in real time. This satisfies the aims of this project since it is more based around researching computer vision techniques rather than attempting to develop an obstacle detection system given very stringent requirements. The specifications of the personal computer used are given below:

Device Specifications

Personal Computer (Compaq Presario)

- Intel Pentium 4 2.8 GHz Processor
- 256 MB RAM
- 40GB HDD
- USB input

Digital Camera (Trust 4400BX)

- Maximum Resolution 2048x1376
- USB output

The digital camera used had a maximum 2048x1376 resolution, however to save processing time while at the same time maintaining the integrity of the system to some extent, all the images were taken with the camera resolution set to 640x480.

3.5 Development Environment

The development environment is vital to the implementation of a system and it must be chosen before commencing the design plan since the design will be influenced by it. Several development environments were considered for this project. C was always one of the preferences, merely because of its popularity more than anything else. Ada had been considered because of its reliability in real-time systems, however it was ruled out since the design does not necessitate a pure real-time system. The most convenient way of representing images in terms of data is in the form of matrices and this project would require very complex manipulations of matrices which most of these languages do not have inbuilt functions for. That is the reason MATLAB was chosen as the environment to develop this system. MATLAB provides the programmer with inbuilt functions for most complex mathematical operations (all that of matrices included) and its image processing toolbox makes it unbelievably simple for the use to process data from images (e.g. edge detection is an inbuilt function) and then represent the processed information in the form of graphs, plots and other images.

3.5.1 The MATLAB Environment

Advantages of MATLAB

- It is an interpreted language which makes it very suitable for development and research purposes since code and data can be updated while still in execution (giving the programmer a sense of interactivity with the system being developed).

- It is a function-based language, which makes it very develop, extend and document programs in.

- It is a language intended for numerical computation. It operates on matrices rather than scalar which simplifies many mathematical operations

- It does not require the user to predefine every variable with a type and a size before its first use.

- It is very easy to exchange data between Matlab and other applications such as Excel which would be an advantage since most numerical analysis performed in the evaluation of this report can be done easily in Excel.

- Its built-in features allow not only the programmer to perform calculations but also present the results visually with minimum complexity in code.

- Matlab allows the programmer to work with undefined values. That is division by zero is allowed without giving an error resulting in a special symbol inf as the answer. Inf can be used in subsequent calculations where its division or multiplication with any other real number results to Inf. This is very useful when working with calculating values that might easily be undefined (e.g. gradients of vertical lines).

Disadvantages of MATLAB

- Since it is an interpreted language, it can be really slow. However, good programming practices can often increase its speed incredibly.

- The fact that variables are not declared can often lead to mistakes made my programmers in spelling variable names etc. to go undetected. Once again, good programming practices would help avoid these problems.

Details of Development Environment Used

MATLAB The Language of Technical Computing
with Image Processing Toolbox
Version 6.5.0.180913a Release3
June 18, 2002

3.5.2 Differences between C and MATLAB

Since C and Matlab were the two main programming environments considered for this project, a very comprehensive study comparing and contrasting the two languages was made.

- C is a compiled language whereas Matlab is an interpreted language. Although a compiled C program will run considerably faster than an interpreted Matlab program, the latter is definitely preferred during development.

- The fact that variables do not have to be declared in Matlab does give the programmer the advantage of fast coding, however unlike in C (where variable declarations are compulsory before use) it also makes the program more susceptible to fail for simple spelling mistakes or typos.

- Although like Matlab, C can also use arrays and matrices, C restricts the size of these data structures on declaration. Furthermore, calculations involving elements of an array would involve large bits of code (with FOR loops) in C, whereas Matlab can perform them with a

simple command (such as matrix multiplication), making it ideal for solving technical problems.

- Many libraries are available for C that give it almost the same functionality as Matlab, however one has to search for these libraries and one can never be entirely sure of the reliability of these libraries. Other than technical calculations Matlab also has inbuilt functions (or purchasable toolboxes) for plotting graphs, linear algebra, differential equations etc.

- In C, all variables declared in one scope are accessible by functions and procedures in the next scopes, however in Matlab all variables need to be explicitly declared at the beginning of a function call to be accessible in the next scope. This can be an advantage since it prevents the programmer from writing code that may inadvertently change the value of a global variable.

- Learning to use the more complicated functions of C would require one to go through a book or other resources. However, Matlab comes very well-documented and it is relatively easy to learn the language even from scratch by simply reading the documentation provided with it.

- Free open-source compilers (such as gcc) are available for C which means that one does not necessarily have to pay for using it, although there are commercial compilers available for it. Matlab on the other hand is a commercial product costing about $100. This was however not an issue since the Department of Computer Science has purchased quite a few licences for students to develop systems in at the department. Some free open-source programming environments such as Octave and Scilab are available that have some of the characteristics and much of the functionality of Matlab.

3.3 Design Plan

The following chart shows each sub-problem in blocks. Blocks further down in the chart are the more basic sub-problems which were solved first while block towards the top of the list are more complex and were solved only after the sub-problems it is dependent on are solved first. Arrows indicate dependency and flow of information between sub-problems, i.e. for sub-problems contributing to the solution of a larger problem there are arrows from the contributing sub-problems to it. These arrows are labelled with the data interchanged between sub-problems and the direction of information flow is indicated by the head of the arrow.

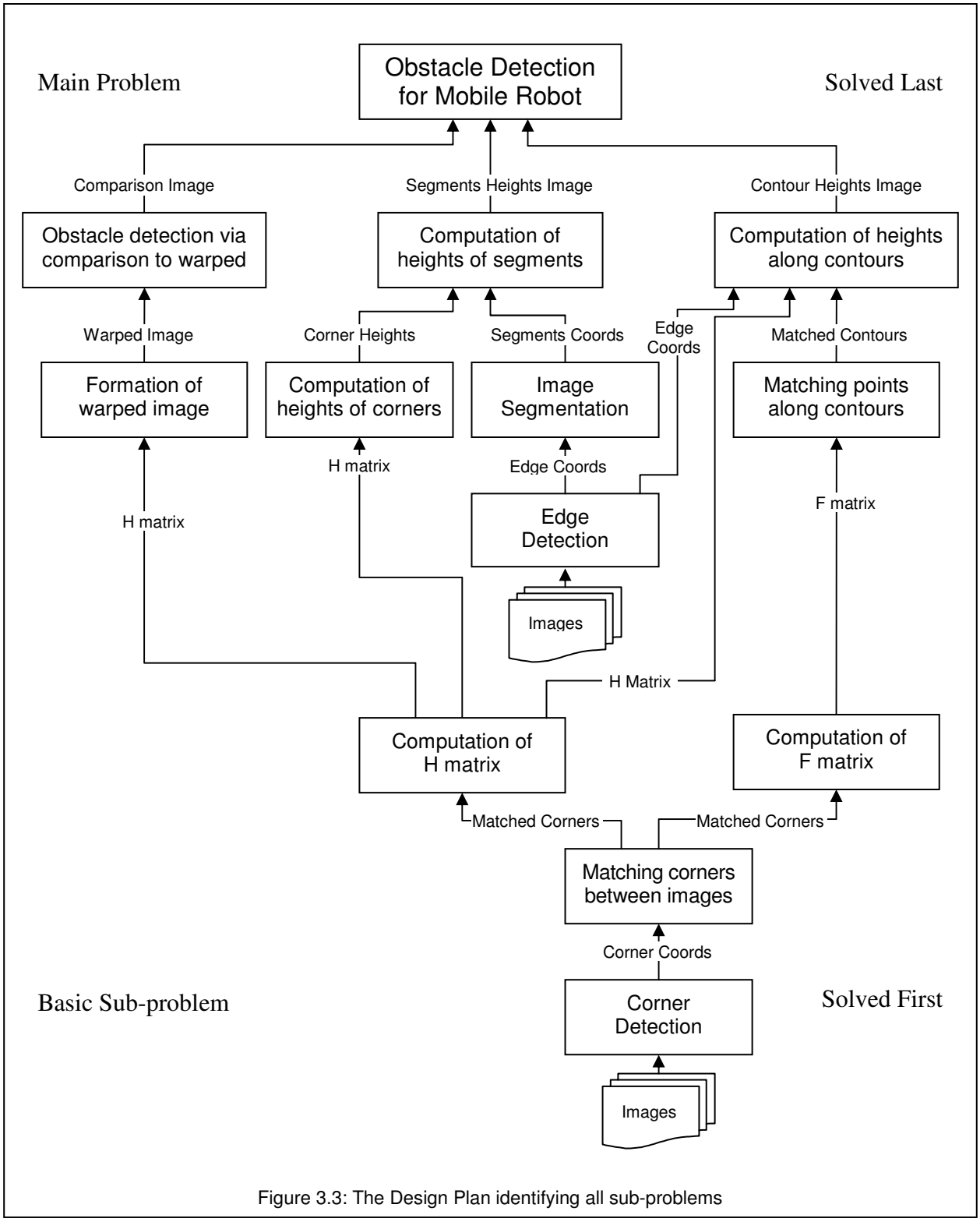


Figure 3.3: The Design Plan identifying all sub-problems

Descriptions of Sub-problems

The following is a brief description of the purpose and planned implementation of the solution to each sub-problem or "module". References have been included to the literature review where appropriate where background knowledge may be required to understand the methods and techniques involved in the solutions. Unless otherwise stated, all the solutions are to be implemented and coded in the development environment by myself.

- ***Corner Detection*** **-** Once the initial and final images have been processed into data using a standard operation of the development environment, corners on both images are located individually. The Harris detector [11] (described in Section 2.6) written by Peter Kovesi [15] is used.

- ***Matching corners between images*** **-** Using the co-ordinates of corners from *Corner Detection*, this module tries to find pairs of corresponding corners between the two images. Its output is a set of pairs of co-ordinates, with each pair being co-ordinates of the same corner on the initial and final image. Correspondences of each point can be found by comparing a set of candidates and selecting the one with the most similar intensities of neighbouring pixels with the one being matched.

- ***Computation of H matrix*** **-** Using the matched corners from *Matching corners between images*, the homography matrix (*H*) is computed for the scene using the RANSAC inhomogenous method (Section 2.4.4)

- ***Computation of F matrix*** **-** Using the matched corners from *Matching corners between images*, the fundamental matrix (*F*) is computed for the scene using the normalised 8 point RANSAC algorithm (Section 2.3.2.2). Code written by Zezhi Chen [5] is used.

- ***Formation of warped image*** **-** The homography matrix from *Computation of H matrix* is used to create a warped image from the initial image. This is basically done by finding the expected positions of all pixels on initial image and carrying their intensities to these computed positions to the warped image (Section 2.4.3). Interpolation is used to "fill in the blanks", i.e. estimate the intensities of pixels not plotted on the warped image.

- ***Obstacle detection via comparison to warped image*** **-** The warped image from *Formation of warped image* is compared to the final image. A new image is formed displaying the difference in intensity of pixels in corresponding locations in the final image and warped image. Furthermore, another image is formed displaying the difference in intensity of blocks (16x16) of pixels in corresponding locations in the final image and warped image. Both these images are indicators of obstacles.

- ***Edge Detection*** **-** The edge detector uses the Canny [4] method described in Section 2.5 to give coordinates of all points that are on edges or contours. Edge detection is a function provided with the MATLAB image processing toolbox incorporating several methods of detection. It also returns a binary image, which indicates whether a pixel at a particular co-ordinate is an edge or not in the initial image. Edge detection is performed both on the initial and final image.

- ***Computation of heights of corners*** **-** Using the homography matrix obtained from *Computation of H Matrix*, the homography matrix of each plane parallel to the ground between the ground and the height of the camera is estimated (Section 2.4.2). Using the matching corner pairs from *Matching corners between images* this module estimates its height in the scene by finding out which homography the pair most closely corresponds to.

- ***Image Segmentation*** **-** Region based segmentation is used (Section 2.7) to segment the final image into several segments or "objects". The result is a matrix of the same dimensions as the final image indicating which segment each point on the final image belongs to. K-means clustering (2.7.1) is performed on the image first followed by segmentation. Code written by Zezhi Chen [5] is to be used for the k-means solution.

- ***Computation of heights of segments*** **-** Using the segmentation information obtained from *Image Segmentation* and the heights of each corner from *Computation of heights of corners* it is possible to compute the height of each segment. This is simply the maximum height among all the corners in each segment. An image is created with high intensity for higher segments and low intensity for segments close to the ground. This is another method of obstacle detection.

- ***Matching points along contours*** **-** Using the fundamental matrix obtained from *Computation of F Matrix* and the coordinates of all points on corners from *Edge Detection*, it is possible to draw the epipolar line for each edge-point (Section 2.3.1). A correspondence for each edge-point on the initial image can be found on the final image by drawing its epipolar line and using it as a search strip for candidate matches. The candidate with neighbouring pixels most similar to the one being matched is chosen as its match.

- ***Computation of heights along contours*** **-** Using the pairs of corresponding points on contours obtained from *Matching points along contours* and the homography matrix obtained from *Computation of H matrix* it possible to find the height of each point on the edge in the scene. The method used is exactly the same as with the *Computation of heights of corners* (Section 2.4.2), the only difference being point correspondences of edge-points are used rather than corner-points. An image is formed with blocks indicating the heights of points on corners (increasing intensity meaning greater altitude). This is basically done by dividing the image in 16*16 blocks and increasing the intensity according the maximum height among the edge points in each block.

- ***Obstacle Detection for a Mobile Robot*** **-** This as the title of this project indicates is the main problem. This merely displays the images from the three obstacle detection methods obtained from *Obstacle detection via comparison to warped image*, *Computation of heights of segments* and *Computation of heights along contours* to the user indicating the present of obstacles in the scene.

3.6 Summary of Design

This chapter has gone through the requirements of this project quite thoroughly and a design plan has been made identifying each sub-problem concerned. Choices have been made on the design methodoly and development tools and methods which have been justified. Given the detailed design plan, the next chapter describes in great detail how each sub-problem is solved in the chosen development environment using techniques and methods explained in Chapter 2 (Literature Review).

4 IMPLEMENTATION

As stated in Chapter 2, the Matlab environment was chosen to develop the system. A bottom up approach was taken to achieve the outcome of this project since it is essentially based on exploiting the use of simple operations that can be put together to perform one complex task in the end. As mentioned in the literature review section of this report, these techniques were drawn from three subfields of computing – image processing, pattern recognition and computer vision. The same sequence as used in the literature review to explain these concepts is followed here to describe the development of these techniques within the chosen environment and within the given constraints and their amalgamation into the system that is the ultimate objective of this project.

4.1 Notes about description of solutions and pseudocodes

- The pseudocodes provided ended up looking much like actual code. This is because most of the pseudocode described complex matrix manipulations which made the its actual code-type appearance inevitable.

- The solution of each sub-problem is described first comprehensively through text followed by pseudocode used to develop the final code. The descriptive text contains references of the form *%ABn* (where *A* and *B* are alphabets and *n* is a number) which correspond to lines on the following pseudocode with the same referencing index. This is there to help the reader associate the description of the solution to each part of the sub-problem with a more low-level description of the solution which is finally used in implementation and coding.

- The common convention is to refer to the position of a point on an image or graph with x-coordinate *x* and y-coordinate *y* simply as (*x,y*). However in Matlab, images are stored in matrices and with matrices the common convention is to refer to elements as *matrix(r,c)* where *r* is the row and *c* is the column at which the element is located. This means that when an image is stored as a matrix (*im*), a point (*x,y*) will be referred to as *im(y,x)*. Obviously this can lead to confusion while cross-referencing between the descriptions of algorithms and their respective pseudocodes. Thus to keep things consistent and to avoid any such confusion, the (x,y)-graph convention is used to refer to pixels on matrices representing images throughout this report, however the (r,c)-matrix convention is used in the actual Matlab code

- Images used are RGB images. For simplicity the values of pixels in images have been referred to as a single element (as would be proper in the case of greyscale images) e.g. as $image(p_x,p_y)$ for pixel *p*, however for a point p on an RGB image $image(p_x,p_y) = (r_p,g_p,b_p)$ where r_a, g_a and b_a are the red, green and blue intensities respectively of the pixel p.

- Additions (or subtractions) performed between a pixel and a number would result in the red, green and blue values of the pixel each being divided by the number resulting in a set of RGB values (i.e. three elements rather than one)

  $image(p_x,p_y)+n = (\{r_p+n\}, \{g_p+n\}, \{b_p+n\})$
  $image(p_x,p_y)-n = (\{r_p-n\}, \{g_p-n\}, \{b_p-n\})$
  $image(p_x,p_y)*n = (\{r_p*n\}, \{g_p*n\}, \{b_p*n\})$
  $image(p_x,p_y)/n = (\{r_p/n\}, \{g_p/n\}, \{b_p/n\})$
  (where n is a number)

  Additions (or subtractions) performed between pixels would result in the red, green and blue values of one pixel being added to (or subtracted from) the red, green and blue values respectively of another pixel, resulting in a set of RGB values. The same idea can be applied to averaging.

  $image(p_x,p_y) + image(q_x,q_y) = (\{r_p+r_q\}, \{g_p+g_q\}, \{b_p+b_q\})$

$image(p_x,p_y) - image(q_x,q_y) = (\{r_p-r_q\}, \{g_p+g_q\},\{ b_p+b_q\})$

$avg\{image(p_x,p_y), image(q_x,q_y), image(s_x,s_y), image(t_x,t_y) \ldots\} = (avg\{r_p,r_q,r_s,r_t,\ldots\}, avg\{g_p,g_q,g_s,g_t,\ldots\}, avg\{b_p,b_q,b_s,b_t,\ldots\})$

However when calculating the sum of differences between a set of pairs of pixels, the result is calculated by first computing the differences of RGB values between the pixels and then summing the three differences. e.g. to calculate sum of differences between pairs of points (p,q), (s,t) and so on:

$sum(diff\{(image(p_x,p_y),image(q_x,q_y)), image(s_x,s_y),image(t_x,t_y)), \ldots.\})) = abs(r_p- r_q)+ abs(r_s- r_t)+\ldots +abs(g_p-g_q)+abs(g_s-g_t)+\ldots + abs(b_p-b_q)+abs(b_s-b_t)+...$

4.2 Point Correspondences

4.2.1 Corner detection

The algorithm commonly known as the Harris corner detector [11] is used to detect corners which are later used to find corresponding points between the images. The code used in this project for corner detection is written by Peter Kovesi [15] from the Department of Computer Science & Software Engineering at the University of Western Australia.

```
[cim, r, c] <- harris(im, sigma, thresh, radius, disp)
```

The function harris takes used the following parameters

im	image to be processed in the form of a matrix
sigma	standard deviation used to perform Gaussian smoothing
thresh	threshold for corner detection (optional)
radius	radius of region considered in non-maximal suppression (optional)
disp	flag which if set the original image is displayed with the corners overlaid on it (optional)

And it returns:

cim	binary image of same dimension as im marking corners, i.e. cim(x,y)=1 if (x,y) is a corner on im
r	matrix of dimensions nx1 with row coordinates of n corner points
c	matrix of dimensions nx1 with column coordinates of n corner points.

The Harris corner detector works by calculating the local structure matrix for each pixel. Then it performs eigen-decomposition of the local structure matrix for each pixel giving two eigenvalues. If the difference between the two eigenvalues is large the pixel is then identified as a corner (Section 2.6)

4.2.2 Matching Corners

As mentioned earlier the Harris corner detector [11] was used to find corners on *image1* and *image2*, with the list of corners finally stored in *n*x2 matrices corners_im1 and corners_im2 respectively. Since the function harris when given an image $image_z$ returns the x and y coordinates of *n* corners in separate (1x*n*) matrices r_z and c_z respectively. The matrix giving a list of corners (with x and y coordinates on each row) $corners\_im_n$ is created simply as one with r_n and c_n as its two columns. Furthermore iscornerim1 and *iscornerim2* are binary matrices of the same dimensions as *image1* and *image2* where $iscornerim_n(x,y)$=1 if (x,y) is a corner on $image_n$ (*%MC0*)

One of the initial requirements to compute the homography of the scene was to find a pairs of corresponding points between the two images. For points to be matched between images they must be unique and distinguishable. Corners are used since they are the most distinguishable points in any image because of their distinct neighbouring regions. It can be assumed that two points are a match if

their neighbouring regions are the same i.e. roughly the difference in RGB/greyscale values between a certain block of neighbouring region around the points are more or less the same.

Let us assume *matched_corners* is a relation consisting of (*x,y*)->(*i,j*) such that (*x,y*) and (*i,j*) are corresponding points between *image1* and *image2* respectively. To find true matches between points *matched_corners* must be a one-one relation.

Therefore for each corner (*x,y*) on image1, corners on *image2* within a certain radius or window (*w*) are selected as potential candidates for a match (*%MC1*). This means that every corner on *image2* with coordinates (*s,t*) where $x-w \leq s \leq x+w$ and $y-w \leq s \leq y+w$ is treated as a candidate match for (*x,y*).

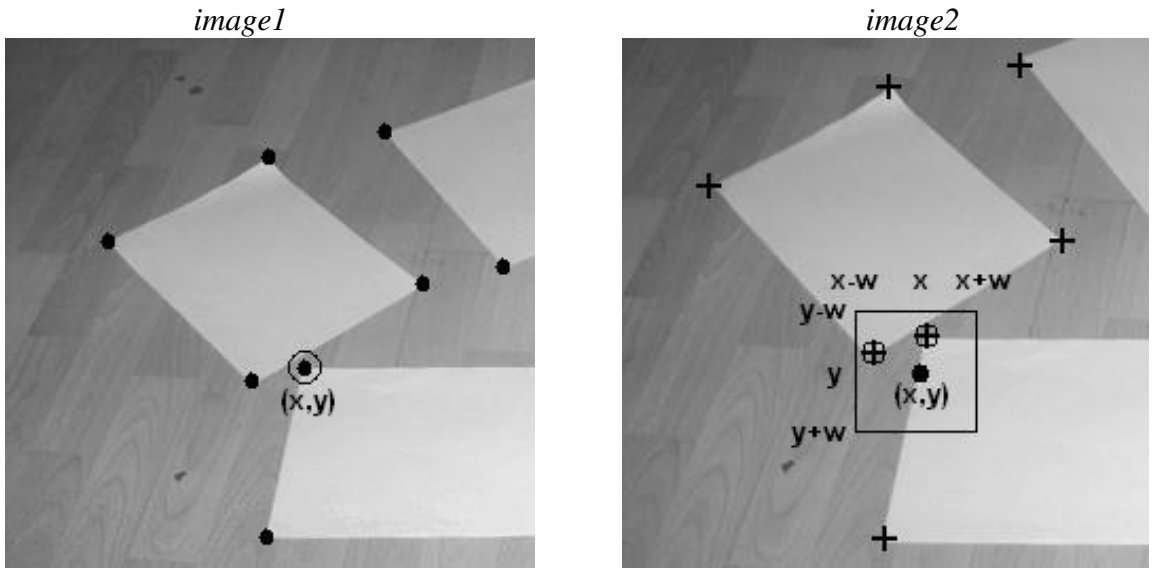


Figure 4.1: Candidate matches for corner correspondences

All corners on *image1* and *image2* are represented as dots and crosses respectively. When finding a point correspondence on *image2* for the corner (*x,y*) on *image1*, all corners on *image2* in the region (*x-w* to *x+w*, *y-w* to *y+w*), i.e. all corners in the square box are considered as candidates. Candidate corners have been marked with circles around the crosses.

It is possible for there to be several corners on *image2* which are close matches that may result to *matched_corners* being a many-many relation (Figure 4.2a). Please note that it is possible for corners on image1 to have no candidate matches on *image2* at all (i.e. if there are no *image2* corners within the window) and similarly for corners on *image2* to qualify as candidate matches for no image1 corners at all.

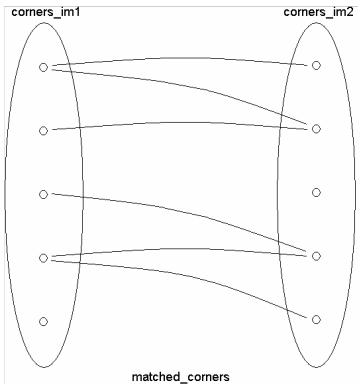


Figure 4.2a: *Matched_corners* as a many-many relation between corners on *image1* and corners on image2 that are its candidate matches

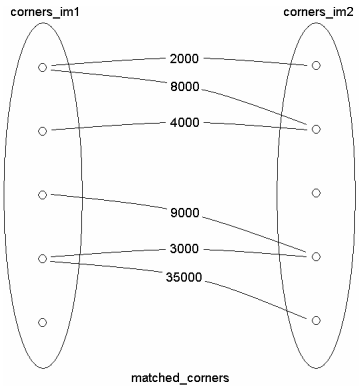
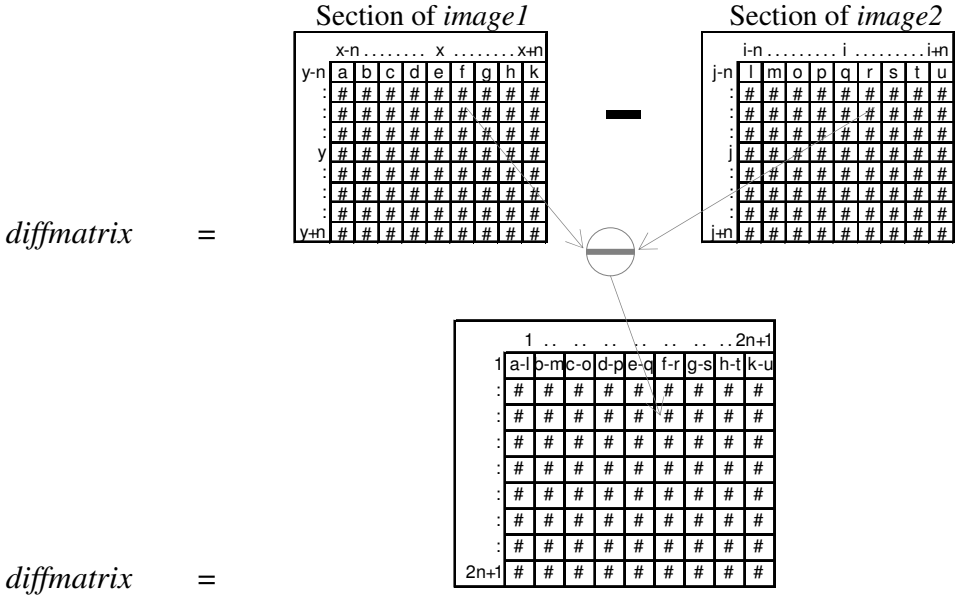


Figure 4.2b: *Matched_corners* as a many-many relation between corners on *image1* and corners on *image2* that are its candidate matches with the link between each pair showing the sum difference between the neighbours of the corresponding pairs

Figure 4.2: Matched corners as a many-many relation

All possible candidates for each corner (*x,y*) on image1 are searched and the difference in neighbouring regions between each corner (*x,y*) on *image1* and each of its candidate matches (*i,j*) on *image2* are calculated. This can be done simply by subtracting the part of matrix *image2* (*i-n* to *i+n*, *j-n* to *j+n*) from part of the matrix *image1* (*x-n* to *x+n*, *y-n* to *y+n*). Then the sum difference would be the sum of the absolute values of all the elements in the matrix resulting from the subtraction (*%MC2*). Please not that in other programming language two for loops would be required to perform the same task as is shown on the pseudocode.



difftotal = sum{abs(*diffmatrix*)}

difftotal = abs(a-l) + abs(b-m) + abs(c-o) + abs(d-p) + ……

Figure 4.2: Calculation of differences in neighbouring regions between pixel (*x,y*) on *image1* and pixel (*i,j*) on *image2* where neighbouring region radius = *n*

If the difference is greater than a certain preset number, then the match is disregarded (*%MC3*). This is simply because if there the difference is too huge then the relationship between the two corners may be considered too unreliable, that is, they are not really matches at all.

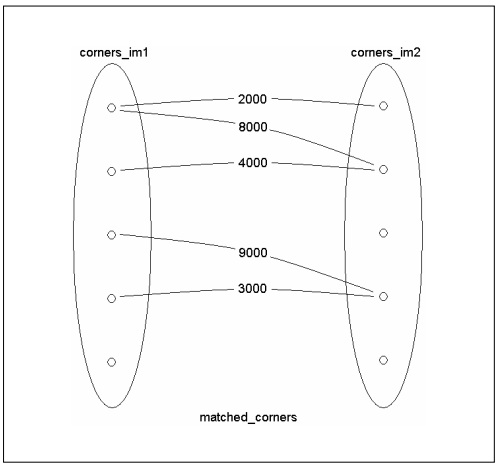


Figure 4.4: *Matched_corners* as a many-many relation between corners on *image1* and corners on *image2* that are its candidate matches with unreliable pairs removed.

The candidate (*i,j*) with the lowest sum difference to (*x,y*) is chosen as a match for (*x,y*) *%MC4*.This means that *matched_corners* is now a many-one relation since it is still possible for several corners on *image1* to match to one corner on *image2*.

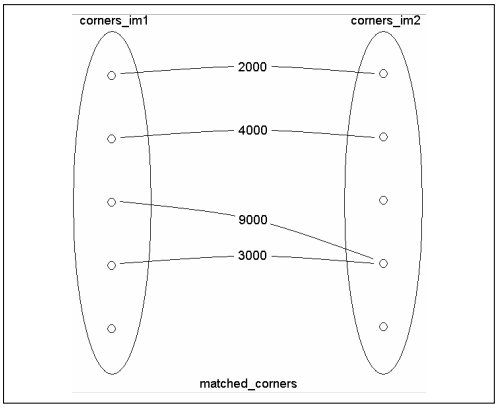


Figure 4.5: *Matched_corners* as a many-one many relation between each corner on *image1* and a corner on *image2* that is its best match.

However matched corners must be a one-one relation to serve the purpose of homography computation efficiently. In order to achieve this, for each (*i,j*) on matched_corners (i.e. each corner on *image2* on matched corners) the entire relation is searched for all corners (*x,y*) on image1 mapping to (*i,j*) and only the corner on *image1* with the least sum difference for neighbours to (*i,j*) is kept whereas the rest are discarded. This means that in case there are multiple matches to any (*i,j*), only the best match is kept finally making *matched_corners* a one-one relation (*%MC5*) as it was aimed to be in order to serve the purposes of this system.

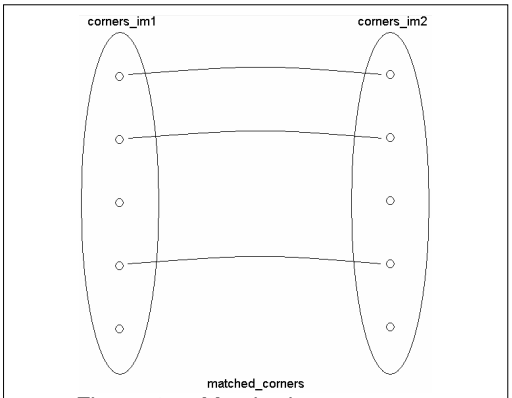


Figure 4.6: *Matched_corners* as a one-one relation with final correspondences

Figure 4.7: Pseudocode for matching corners

```
[iscornerim1, r1, c1] <- harris(image1, 2)                          %MC0
corners_im1 = [r1 c1]
[iscornerim2, r1, c1] <- harris(image2, 2)
corners_im2 = [r2 c2]

cornertotals = ()
for each row (x,y) on corners_im1
      mintotal = 30000                                              %MC3
      matchfound = 0
      for i = x-w to x+w
            for j = y-w to y+w
                  if (i,j) is a member of corners_im2 then          %MC1
                        difftotal = 0

                        difftotal = sum( abs{
                          image1(x-nhood:x+nhood, y-nhood:y+nhood) -
                          image2(i-nhood:i+nhood, j-nhood:j+nhood)} )
                                                                    %MC2
                        if difftotal<mintotal then                  %MC4
                              mintotal = difftotal
                              tempi = i
                              tempj = j
                              found = 1
                        end
                  end
            end
      end
      if found = 1 then                                             %MC4
            add (x, y, tempi, tempj, difftotal) to cornertotals
      end
end

matched_corners = ()
for each row (x, y, i, j, dtotal) in cornertotals
      mintotal = 65535
      for each row (a, b, p, q, dtotal)
            if (p, q) = (i, j) and dtotal < mintotal                %MC5
                  tempx = a
                  tempy = b
                  mintotal = dtotal
            end
      end
      add (tempx, tempy, i, j) to matched_corners
end
```

4.3 Homography Computation

The aim is to compute the homography matrix for the ground plane between the two images *image1* and *image2*. Since H has 8 degrees of freedom, it can be computed using four pairs of matching points between image1 and image2.

Some corners are initially discarded as potentials for calculating H (*%HC1*). These include pairs of corners which have too little difference between their positions (<5 in Euclidian distance) and corners in the upper half of the image (since it can be assumed that corners in the lower half are most likely to be ground corners). The algorithm used is a variation of the inhomogenous RANSAC computation of H (Section 2.4.4) developed by myself. The only difference is instead of selecting the homography with the most inliers, the homography with the highest degree of correspondence to corners is selected.

The following process is repeated several times, each time calculating a new value of H (*%HC2*).

Four random corners are selected first (i.e. four rows from *matched_corners* are picked) and then it is made certain that no three of these four corners lie on the same line (*%HC3*). This can be done by checking whether for each trio of corners out of the four (i.e. 4 combinations of 3 corners each) the gradient between two pairs of points from the trio are the same. Then with this set of four corners the h matrix is calculated. The next step is to find out how good the corners selected were for estimating the homography of the ground plane. This can be done by comparing how good estimates of warped positions the homography matrix gives for each corner.

For a corner on the initial image (*ix,iy*) and its corresponding corner on the final image (*fx,fy*), its warped (expected) position (*ex,ey*) is computed using

$(ex,ey,tmp)^t = h\_matrix*(ix,iy,1)^t$

A comparison is made between ex and ey using the following variables (*%HC4*).
p = Euclidian distance between (*ix,iy*) and (*ex,ey*)
q = Euclidian distance between (*ix,iy*) and (*fx,fy*)
r = Euclidian distance between (*fx,fy*) and (*ex,ey*)
t = angle between lines (*ix,iy*)to(*ex,ey*) and (*ix,iy*)to(*fx,fy*), using cosine formula on *p*, *q* and *r*.
$d = t + r*100/p$
The variable d represents a comparison between the final corner and warped corner. Small values of d indicate that the point d correspond to the same homography as H. This method of comparing final points and warped points will be referred to as the d method throughout the rest of this report.
The sum of *d*'s (*diff*) is stored for each *H* (*%HC5*).

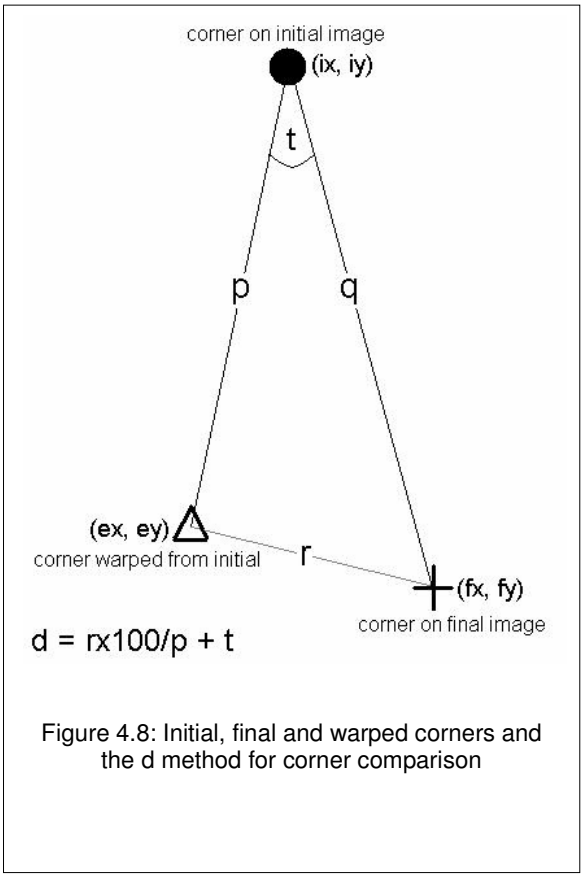


Figure 4.8: Initial, final and warped corners and the d method for corner comparison

The correspondence *d* between the final and warped corner is thus calculated as a sum of angle between them (from the initial corner) and the distance between them expressed as a percentage of the distance between the initial and final corner (since in the case of small values of *p*, a simple difference *r* would not be very reliable). Large values of d, indicate that the corner does not correspond to the homography used.

Please note that on the final output corners on the initial image are shown as circles, those on the final image are shown as crosses and warped corners are shown as triangles. The lines *p* and *q* (but not *r*) are shown for each trio of such corners.

After calculating several values of *H* and their corresponding *diff*'s, the value of *H* with the minimum corresponding value of diff (i.e. the *H* which most corners seem to agree with) is chosen as *h_matrix* (*%HC6*).

Figure 4.9: Pseudocode for Homography Computation (Part1)

```
h_corners = ()
for each row (x, y, s, t) in matched_corners                          %HC1
      if (x-s)^2+(s-t)^2 > 25 & x > rows/2 then
            add (x,y,i,j) to h_corners
      end
end


mintotald = 65535
for n = 1 to 100                                                      %HC2

      repeat
            randomly pick 4 elements (x_i,y_i,s_i,t_i) from h_corners
      until any 3 (x_i,y_i)'s do not lie on the same line             %HC3

      totald = 0

      temp1 =     -x1   -y1   -1     0     0     0    s1*x1   s1*y1
                   0     0     0    -x1   -y1   -1    t1*x1   t1*y1
                  -x2   -y2   -1     0     0     0    s2*x2   s2*y2
                   0     0     0    -x2   -y2   -1    t2*x2   t2*y2
                  -x3   -y3   -1     0     0     0    s3*x3   s3*y3
                   0     0     0    -x3   -y3   -1    t3*x3   t3*y3
                  -x4   -y4   -1     0     0     0    s4*x4   s4*y4
                   0     0     0    -x4   -y4   -1    t4*x4   t4*y4

      temp2 =     -s1
                  -t1
                  -s2
                  -t2
                  -s3
                  -t3
                  -s4
                  -t4

      hmp = inv(temp1)*temp2

      tmphmatrix = hmp(1) hmp(2) hmp(3)
                   hmp(4) hmp(5) hmp(6)
                   hmp(7) hmp(8)   1

      for each row(ix,iy,fx,fy) in matched_corners
            (ex,ey,tmp)^t = h_matrix*(ix,iy,1)^t                      %HC4
            p = {(ix-ex)^2 + (iy-ey)^2}^½;
            q = {(fx-ix)^2 + (fy-iy)^2}^½;
            r = {(fx-ex)^2 + (fy-ey)^2}^½;
            t = (arccos{(p^2 + q^2 - r^2) / (2*p*q)}) * 180/pi;
            d = r*100/q + t
```

```
            totald = totald + d                                          %HC5
      end

      if totald < mintotald then                                         %HC6
            h_matrix = tmphmatrix
            mintotald = totald
      end
end


relcorners = ()
for each row(ix,iy,fx,fy) in matched_corners

      (ex,ey,tmp)^t = h_matrix*(ix,iy,1)^t

      p = {(ix-ex)^2 + (iy-ey)^2}^½;
      q = {(fx-ix)^2 + (fy-iy)^2}^½;
      r = {(fx-ex)^2 + (fy-ey)^2}^½;
      t = (arccos{(p^2 + q^2 - r^2) / (2*p*q)}) * 180/pi;
      d = r*100/q + t

      if t<25 & d<40 & r<25                                              %HC7
            add (ix,iy,fx,fy) to relcorners
      end

end
```

Then corners within a certain threshold of *d* used by calculating *H* are selected by calculating *d* similarly as done earlier (*%HC7*). The selected corners are now used to compute an even better estimate of *H* (*%HC8*), which become the final homography matrix (for the ground plane) used throughout the rest of the program for *image1* and *image2*.

Afterwards the expected warped positions of each corner from *image1* in *matched_corners* are estimated using *h_matrix*. Then the coordinates of the corner on the initial image, its corresponding corner on the final image and the expected corner (found from warping using *h_matrix*) are stored in rows of a matrix *cornerswrp* (*%HC9*). Furthermore, the initial, final and expected corners are plotted as dots, crosses and triangles respectively on both images with line between each pair of initial and final corners and another line between each pair of initial and expected corners. (*%HC10*)

Figure 4.10: Pseudocode for Homography Computation (Part2)

```
temp1 = ()                                                               %HC8
temp2 = ()
for each row (x,y,s,t) in relcorners

      add        -x     -y    -1     0     0     0    s*x    s*y
                  0      0     0    -x    -y    -1    t*x    t*y
      to temp1

      add         -s
                  -t
      to temp2

end

hmp = inv(temp1)*temp2

h_matrix =   hmp(1) hmp(2) hmp(3)
```

```
            hmp(4) hmp(5) hmp(6)
            hmp(7) hmp(8)   1

cornerswrp = ()
display(image1)
display(image2)
for each row (ix,iy,fx,fy) in matched_corners

      (ex,ey,tmp)^t = h_matrix*(ix,iy,1)^t                          %HC9
      add (ix,iy,fx,fy,ex,ey) to cornerswrp

      plot (ix,iy) as dot on image1                                 %HC10
      plot (fx,fy) as cross on image1
      plot (ex,ey) as triangle on image1
      plot line between (ix,iy) and (fx,fy) on image1
      plot line between (ix,iy) and (ex,ey) on image1

      plot (ix,iy) as dot on image2
      plot (fx,fy) as cross on image2
      plot (ex,ey) as triangle on image2
      plot line between (ix,iy) and (fx,fy) on image2
      plot line between (ix,iy) and (ex,ey) on image2

end
```

4.4 Warping Images

In order to detect obstacles, the first step taken is to the warped image from the homography estimated earlier and the initial image (Section 2.4.3). This is basically an image of the scene from the same camera position as where the final image *image2* was taken IF all the pixels in *image1* were located on the ground plane (since the homography estimated is assumed to correspond to the ground plane).

For each pixel (*x,y*) on image1, its position (*s,t*) on *imagew* can be calculated using the homography relationship (*%IW1*)

`(s t tmp)^t = h_matrix*(x y 1)^t`

Then by executing *imagew(s,t) = image1(x,y)* the contents of the pixel are transferred to *imagew*.
However, this method is not complete since it is likely that some pixels on *imagew* may not be mapped on to at all and some pixels may be mapped more than once.

Let us first consider the case of the latter. Another matrix *plottedw* of same dimensions as *imagew* is initialised with 0s, such that *plottedw(s,t)* indicates how many (*x,y*)'s have mapped on to (*s,t*) on *imagew*. Everytime a point (*s,t*) on *imagew* is plotted on to by a point(*x,y*) in *image1*, *plottedw(s,t)* is incremented by 1 and the contents of *image1(x,y)* is added to *imagec(s,t)*. This means that finally *plottedw(s,t)* holds the number of times each point (*s,t*) on *imagew* has been plotted and *imagew(s,t)* hold the sum of all values of *image1(x,y)* that (*s,t*) has been plotted on to by (*%IW2*). In order for *imagew* to hold the actual warped image, the average value for each pixel is found. This can be done simply by dividing each element of *imagew* (for pixels that have been plotted at least once) by the corresponding element in *plottedw* (*%IW3*).

Now let us the consider the case where a point (*s,t*) on *imagew* has not been plotted at all. The simplest thing to do would be average the values of all its eight neighbouring pixels and store that in *imagew(s,t).*

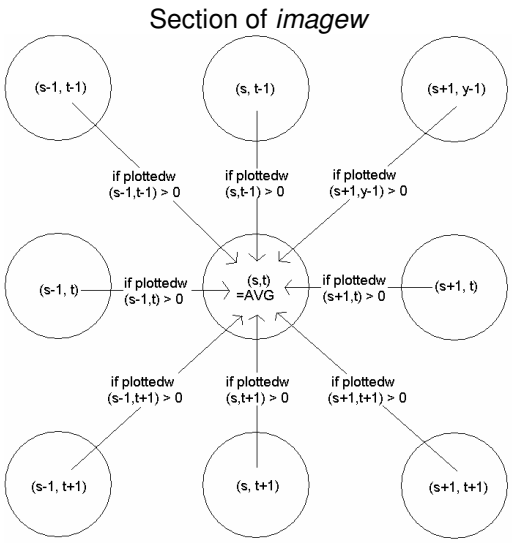


Figure 4.11: The algorithm used for interpolating the value of each pixel (*s,t*) averaging that of its neighbours that have been plotted already

However one must also consider the possibility that some of its neighbouring pixels may not have been plotted either. The basic idea is to find out the number of (*s,t*)'s neighbouring pixels that are plotted *nbrs_pltd* by checking whether *plottedw(i,j)* > 0 and summing the value of image w(*i,j*)'s where (*i,j*) represent neighbours that are plotted already *nbrs_sumd (%IW4*). The value of *imagec(s,t)* can then be interpolated by averaging these values
i.e. *imagec(s,t) = nbrs_sumd / nbrs_pltd*
and then *plotted3(s,t)* is set to 1 indicating (*s,t*) has now been plotted (*%IW5*)
In case none of the neighbours have been plotted, the pixel is left as it is while a count is kept (*%IW6*) of how many pixels have not been plotted at all (*notplotted*).

The entire process is repeated until all the pixels on *imagec* have been plotted i.e. until *notplotted*=0 *(%IW7*). It is a safe assumption that this loop will terminate given that at least one pixel on *imagec* has been plotted and the rest would be interpolated from it.

Figure 4.12: Psuedocode for Warping Images

```
imagew = rows * columns matrix 0s
plottedw = rows * columns matrix 0s

for x = 1 to rows
      for y = 1 to columns
            (s,t,tmp)^t = h_matrix*(x,y,1)^t                    %IW1
            if s>=1 & t>=1 & s<=rows & t<=columns then          %IW2
                  imagew(s,t) = imagew(s,t) + image1(x,y)
                  plottedw(s,t)= plottedw(s,t) + 1
            end
      end
end

for s = 1 to rows
      for t = 1 to columns
            if plottedw(s,t)>0 then
                  imagew(s,t) = imagew(s,t)/plottedw(s,t)       %IW3
            end
      end
end
```

```
repeat
    notplotted = 0
    for x = 1 to rows
      for y = 1 to columns
            nbrs_pltd = 0
            nbrs_sumd = 0
            if plottedw(x,y) = 0 then                                    %IW4

                  if plottedw(x-1,y-1)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x-1,y-1)
                        nbrs_pltd = nbrs_pltd + 1
                  end
                  if plottedw(x,y-1)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x,y-1)
                        nbrs_pltd = nbrs_pltd + 1
                  end
                  if plottedw(x+1,y-1)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x+1,y-1)
                        nbrs_pltd = nbrs_pltd + 1
                  end
                  if plottedw(x+1,y)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x+1,y)
                        nbrs_pltd = nbrs_pltd + 1
                  end
                  if plottedw(x+1,y+1)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x+1,y+1)
                        nbrs_pltd = nbrs_pltd + 1
                  end
                  if plottedw(x,y+1)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x,y+1)
                        nbrs_pltd = nbrs_pltd + 1
                  end
                  if plottedw(x-1,y+1)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x-1,y+1)
                        nbrs_pltd = nbrs_pltd + 1
                  end
                  if plottedw(x-1,y+1)>0 then
                        nbrs_sumd = nbrs_sumd + imagew(x-1,y+1)
                        nbrs_pltd = nbrs_pltd + 1
                  end

                  if nbrs_sumd > 0                                       %IW5
                        plottedw(x,y) = 1
                        imagew(x,y) = nbrs_sumd/nbrs_pltd
                  else
                        notplotted = notplotted+1                        %IW6
                  end

            end
      end
    end
until notplotted=0                                                       %IW7

display(imagew)
```

4.4.1 Warp Comparator

Perhaps the crudest method of obstacle detection is from a comparison between the warped image *imagew* found earlier and the final image *image2*. A pixel-pixel comparison would be pointless since the likelihood of every pixel on the ground plane matching perfectly regardless of the accuracy of our estimation of h is rather minuscule. The approach taken is to compare 16x16 blocks of *image2* with corresponding 16x16 blocks of *imagew*, calculating the difference between each corresponding pixel and summing their absolute values to give the block differences (*blockdiff*). If *blockdiff* for a particular block is above a certain threshold, it is safe to assume the pixel on that block do not correspond well with the homography of the ground plane and the entire block can thus be treated as an obstacle.

As mentioned earlier both images are divided into blocks of 16*16 pixels.
So, block *(i,j)* for $image_x$ would encompass pixels ({*i*-1}*16+1 to *i**16, {*j*-1}*16+1 to *j**16) from $image_x$, i.e. a 16x16 matrix of pixels. e.g. block *(i*=1,*j*=1)* would include pixels with x-coordinates 1 to 16 and y-coordinates 1 to 16, i.e. 256 pixels. The block difference *blockdiff(i,j)* for each block (*i,j*) is calculated simply by subtracting the parts of matrices from *imagew* and *image2* that block (*i,j*) encapsulates and then summing their absolute values (*%WC1*).

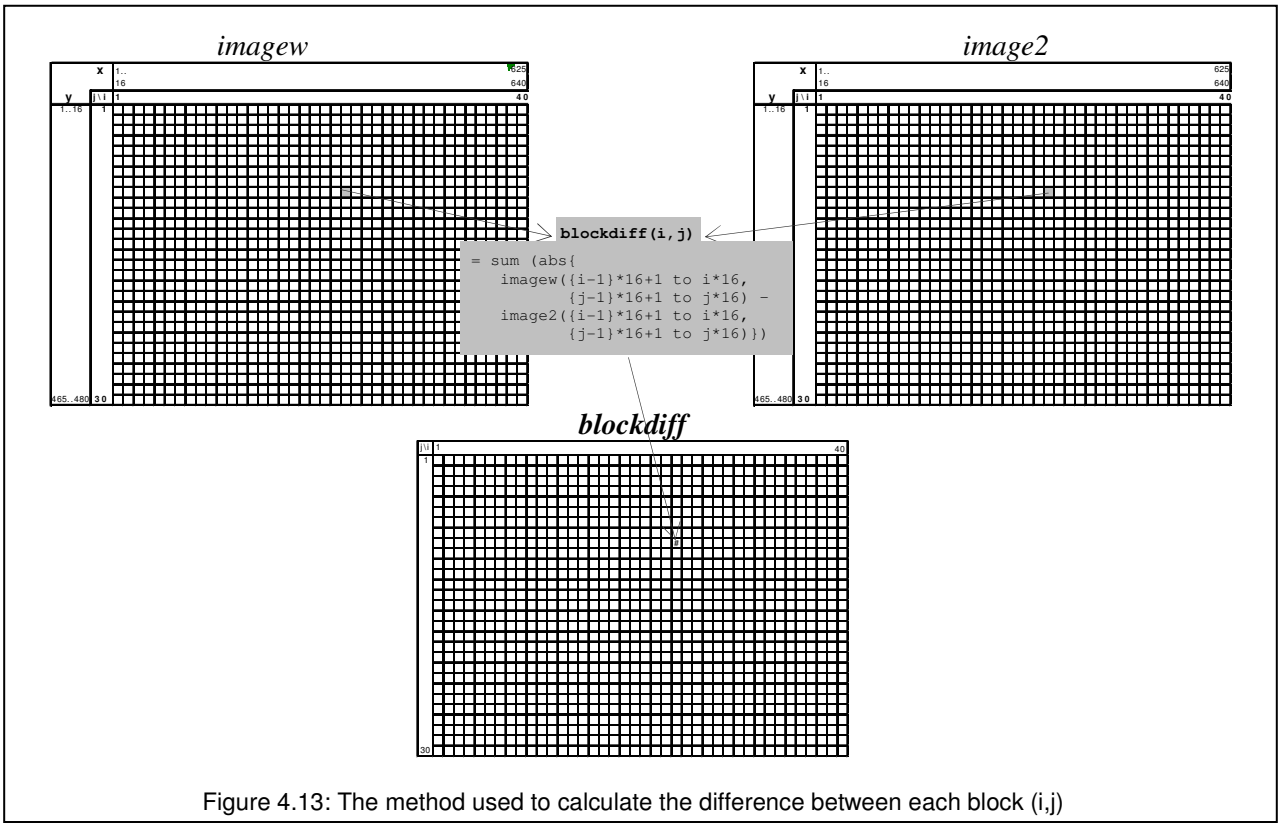


Figure 4.13: The method used to calculate the difference between each block (i,j)

Therefore *blockdiff* would contain the differences between corresponding pixels from each block in *image2* and *imagew*. As stated earlier, if *blockdiff(i,j)* is above a certain threshold all pixels in block (*i,j*) are considered parts of obstacles (*%WC2*).

An improvement is made to this approach to obstacle detection by taking into account corners that lie in each block. If any corner in a block (*i,j*) does not correspond to the homography of the ground plane by using the d method (*%WC3*) i.e. if d is above a certain tolerance limit, it can be assumed that it is part of an obstacle and the entire block can be marked as one. This is done simply by adding the threshold to *blockdiff(i,j)* if (*i,j*) contains any such corners. This means that regardless of the block

differences previously found earlier, the discovery of a non-corresponding corner in block (*i,j*) would force *blockdiff(i,j)* across the threshold indicating an obstacle (*%WC4*).

Finally a new image *imagec* is created representing the comparison between the final image *image2* and the warped image *imagew*. *Imagec* is essentially a copy of image2, however blocks on *imagec* that have *blockdiff(i,j)*>threshold are left as black indicating obstacles. This can be done by initialising *imagec* as composed of just black (0 value) pixels and then copying all pixels from all blocks (*i,j*) from *image2* to *imagec* unless *blockdiff(i,j)*>threshold (*%WC2*).

The image *imagec* can then just be displayed as any standard image is displayed while blocks of black on the display will indicate obstacles found using the warping method (*%WC5*)

Figure 4.14: Pseudocode for Warp Comparator

```
blockdiff = 30*40 matrix of 0s
imagec = 480*640 matrix of 0s

for each row(ix,iy,fx,fy) in matched_corners

      (ex,ey,tmp)^t = h_matrix*(ix,iy,1)^t                          %WC3

      p = {(ix-ex)^2 + (iy-ey)^2}^½;
      q = {(fx-ix)^2 + (fy-iy)^2}^½;
      r = {(fx-ex)^2 + (fy-ey)^2}^½;
      t = (arccos{(p^2 + q^2 - r^2) / (2*p*q)}) * 180/pi;
      d = r*100/q + t

      if t>25 & r>25                                                %WC4
            blockdiff(
             (ex-mod(ex,16))/16+1, (ey-mod(ey,16))/16+1) = 10000
      end

end


for i = 1 to 30
      for j = 1 to 40

            for x = (i-1)*16+1 to i*16
                  for y = (j-1)*16+1 to j*16
                        blockdiff(i,j) = blockdiff(i,j) +
                              abs{imagew(x,y)-image2(x,y)}
                  end
            end

                                                                    %WC1
            blockdiff(i,j)= sum (abs {
                              imagew({i-1}*16+1:i*16,{j-1}*16+1:j*16)-
                              image2({i-1}*16+1:i*16,{j-1}*16+1:j*16)})


            if blockdiff(i,j)<10000                                 %WC2
                  imagec((i-1)*16+1 to i*16, (j-1)*16+1 to j*16)
                  = imagew((i-1)*16+1 to i*16, (j-1)*16+1 to j*16)
            end if
      end
end

display(imagec)                                                     %WC5
```

4.5 Fundamental Matrix Computation

Due to time and other constraints, the code used to compute the fundamental matrix for the scene was not written by myself. Code for fundamental matrix computation written by Zezhi Chen [5] who is a research associate at The Department of Computer Science, University of York (currently working in computer vision) is used with his explicit permission.

The algorithm, given pairs of matching points, uses RANSAC to compute the fundamental matrix which it returns along with a set of pairs of matching points that agree well with the epipolar geometry computed.

```
[matrix, goodmatchL, goodmatchR, residual] <-
fundamental_RANSAC(matchL,matchR)
```

The function takes in *matchL* and *matchR* (which are 3xn matrices) as parameters which contains homogenous co-ordinates of n points on each column in *matchL* and its match on the corresponding column is in *matchR*.

Similarly *goodmatchL* and *goodmatchR* are 3xn matrices containing homogenous co-ordinates of matching points given that these points agree well with the fundamental matrix computed.

The returned value *residual* is the average residual of good matching pairs.

The algorithm is the 8-point normalised algorithm for the computation of F (Section 2.3.2.2). After the normalisation of point-correspondences given as parameters, it uses RANSAC to randomly select a subset of these correspondences and computes the fundamental matrix. It then computes epipolar lines for each point on the initial (or left) image and then calculates the distance of the their matches on the final (or right) image to it. Point correspondences for which this distance is below a threshold are considered as inliers. The process of random selection, matrix computation and inlier identification is repeated several times and the fundamental matrix with the most number of inliers is selected for further computation. The fundamental matrix is then recomputed using all the points that were earlier identified as inliers.

4.6 Image Segmentation and Edge Detection

4.6.1 Colour Segmentation and Edge Detection

Code written by Chen [5] is also used to segment images and find edges based on colour variation. His algorithm uses the pattern recognition technique of k-means clustering (Section 2.7.1) to classify pixels on the image according to colour thus performing colour segmentation. After that the standard Matlab function *edge* is used (with "Canny" as the chosen method) to perform edge detection.

```
[colour_seg  edge_seg]<- kmean_seg(im)
```

When given an image *im*, it returns the following images (or matrices) which are of the same dimensions as *im*

colour_seg - This is the simply the image *im* after the k-means clustering method is performed, that is, the pixels on the image have been put into distinct classes depending on their colour.

edge_seg - This is the result of performing canny edge-detection on the image *colour_seg*. It is a binary binary image with 1s representing edges and 0s representing any other pixels on *colour_seg*.

So, if the point (x,y) was part of an edge on the initial image *im* then $edge\_seg(x,y)=1$ otherwise $edge\_seg(x,y)=0$

4.6.2 Object Segmentation

Chen's algorithm [5] for colour segmentation returns an image *imageseg* which basically used k-means clustering for the classification of pixels that are similar in colour. However this means that pixels on two separate objects of similar colour will be members of the same class. The idea here is to classify the pixels such that only pixels on the same object (with an enclosed edge) are members of the same class.

A very simple approach is taken to accomplish this from the already colour segmented image *imageseg*. The method involves taking each pixel and then finding out which of its already classified neighbouring pixels it matches most closely with. Since pixels are selected left to right top to bottom, the already classified neighbouring pixels would be its left, top and top left neighbours. The pixel (x,y) is classified into the same segment as that of the neighbour (i,j) it most closely resembles in terms of colour segmentation (i.e. has minimum difference in value) by setting $segment(x,y) = segment(i,j)$ %OS1. However if this minimum difference is above a threshold, we can assume it is part of a different segment. In such cases, a new segment/class is created and the pixel (x,y) is assigned to it by setting segment(x,y) = newsegment (%OS2).

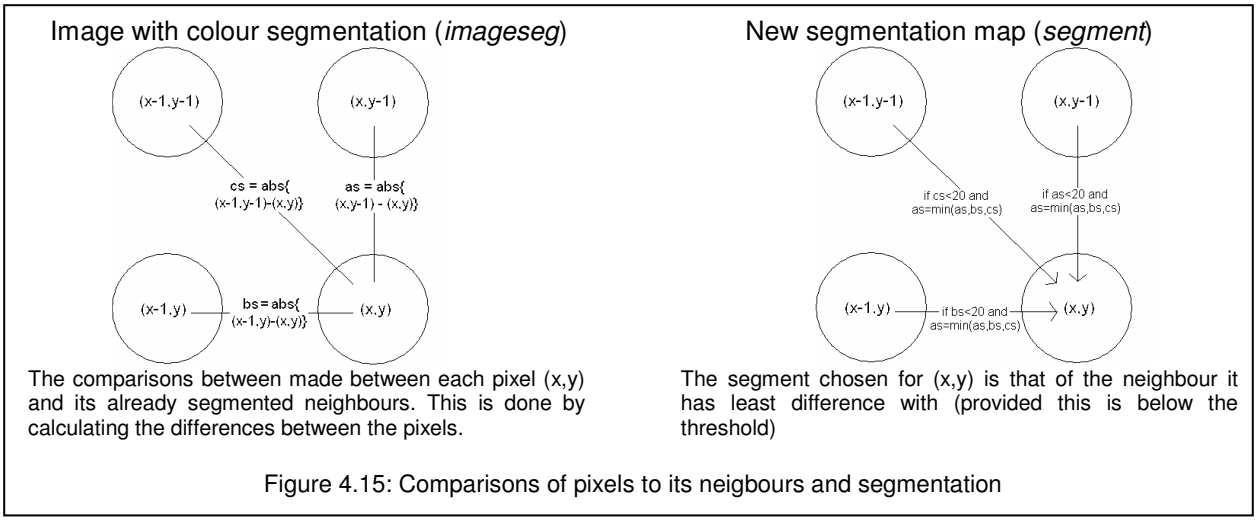


Figure 4.15: Comparisons of pixels to its neigbours and segmentation

Please note that for pixels on the first row or first column, there is only one neighbouring pixel that has been classified so there is no need to find a minimum (%OS3). The first pixel (1,1) is set as part of segment 1, i.e. $segment(1,1)=1$, to initialise the process (%OS4).

It is still possible for an object to be divided into several segments. This may happen if parts of the object were approached at different times by the segmenter algorithm and a new segment was created each of these times. To tackle with this problem, all pairs of consecutive pixels must be analysed and if they are similar in intensity, the segment of the second pixel must be set to the segment of the first pixel. Moreover, all the pixels earlier in the segment of the second pixel are assigned the same segment as the first pixel.

Figure 4.16: Pseudocode for segmentation

```
segment = rows * columns matrix, 0s
nextsegment = 2
segment(1,1) = 1                                                    %OS4

for i = 2 to columns                                                %OS3
      if im(1,i) - im(1,i-1) <= 10
            segment(1,i) = segment(1,i-1)
      else
            segment(1,i) = nextsegment
            nextsegment = nextsegment+1
      end
end

for i = 2 to rows                                                   %OS3
      if im(i,1) - im(i-1,1) <= 10
            segment(i,1) = segment(i-1,1)
      else
            segment(i,1) = nextsegment
            nextsegment = nextsegment + 1
      end
end

for i = 2 to rows
      for j = 2 to columns
            as = abs(im(i,j) - (im(i,j-1)))
            bs = abs(im(i,j) - (im(i-1, j)))
            cs = abs(im(i,j) - (im(i-1,j-1)))

            if min(as,bs,cs) <= 20                                  %OS1
                  if as <= bs & as <= cs
                    segments(i,j) = segments(i,j-1)
                  else
                    if bs <= as & bs <= cs
                      segments(i,j) = segments(i-1,j)
                    else
                      segments(i,j) = segments(i-1,j-1)
                    end
                  end
            else
                  segments(i,j) = nextsegment                       %OS2
                  nextsegment = nextsegment + 1
            end
        end

    end
end


for i = 2 to rows
      for j = 2 to columns
            as = abs(im(i,j) - (im(i,j-1)))
            bs = abs(im(i,j) - (im(i-1, j)))
            cs = abs(im(i,j) - (im(i-1,j-1)))

            if as = min(as,bs,cs) then (nx, ny) = (i, j-1)
            if bs = min(as,bs,cs) then (nx, ny) = (i-1, j)
            if cs = min(as,bs,cs) then (nx, ny) = (i-1, j-1)
                                                                    %OS1
            if min(as,bs,cs) <= 20  and segments(nx,ny) /= segments(i,j)
```

```
                replacedseg = segments(nx,ny)

                for u = 1 to rows
                      for v = 1 to columns
                            if segments(u,v) = replacedseg then
                                  segments(u,v)=segments(i,j)
                            end
                      end
                end

          end
    end
end
```

4.7 Height estimation

4.7.1 Height estimation of points

In order to estimate the height of each point, it is necessary to know the homography of the plane it corresponds to (Section 2.4.2). So, homographies of each plane at intervals of one distance unit (cm) are computed using the standard homography formula for parallel planes using *h_matrix* as H from distance = camera height (*ch*) and *h_inf* as H from distance = infinity (*%HP1*). *H_matrices* is assumed to be a 3 dimensional array of dimensions (0-*ch*,1-3,1-3). It is computed such that *h_matrices*(*i*,1-3,1-3) returns a 3*3 matrix that is the homography of the plane at height *i*. *h_inf* is assumed to be the identity matrix for simplicity in this project since points at infinity hardly seem to move regardless of camera motion

Figure 4.17: Pseudocode for homographies of parallel planes

```
h_matrices = ()                                                    %HP1
h_inf = 1  0  0
        0  1  0
        0  0  1

for i = 0 to ch
      ht = ch - i
      h_matrices(i,1:3,1:3) = h_matrix*(ch/ht) + h_inf*(1-ch/ht)
end
```

In order to estimate the height of a point (*ix,iy*) on image1 which has a corresponding point (fx,fy) on image2, (*ex,ey*) is calculated for (*ix,iy*) for each parallel plane at heights *i* = 0 to *ch* and a comparison is made between (*ex,ey*) and (*fx,fy*) using the d method (explained earlier) *%HP2*. The height *i* selected as the estimated height for (*ix,iy*) is the height at which using *h_matrices*(*i*) the comparing variable d has the lowest value, i.e. the height at which the point (*ix,iy*) seems to match the best to (*%HP3*). Please note that since Matlab does not allow global variables *h_matrices* needs to passed as a parameter for *calculateheight*.

```
height <- calculateheight(ix,iy,fx,fy,h_matrices
```

Figure 4.18: Pseudocode for height estimation of points

```
mindiff = 255
height = 0

for i = 0 to ch
      (ex,ey,tmp)^t = h_matrices(i,1:3,1:3)*(ix,iy,1)^t                %HP2
      p = {(ix-ex)^2 + (iy-ey)^2}^½
      q = {(fx-ix)^2 + (fy-iy)^2}^½
      r = {(fx-ex)^2 + (fy-ey)^2}^½
      t = (arccos{(p^2 + q^2 - r^2) / (2*p*q)}) * 180/pi
      d = r*100/q + t

      if d<mindiff then                                                %HP3
            height = i
            diff = d
      end
end

return height
```

4.7.2 Height estimation for segments

The height of each segment can be found simply by going through each corner and then calculating the heights of each corner. If the height *h* of a corner (*x,y*) is greater than the height the currently set height for the segment it belongs to i.e. *segment*(*x,y*), *h* is set as the new height of that particular segment (*%HS1*). This way, the maximum height among the corners of a segment is stored as the height of the segment.

Figure 4.19: Pseudocode for height estimation of segments

```
segheights = (nextsegment-1)*1 matrix 0s

for each row (x,y,s,t) in matched_corners
      h = height(x,y,s,t,h_matrices)
      if h > segheight(segment(x,y))                                   %HS1
            segheight(segment(x,y)) = h
      end
end

heightsegs = rows * columns matrix 0s
```

Finally an image *heightimage* is created showing the heights of each segment. This is done by assigning each pixel (*x,y*) on *heightimage* as the height of the segment(*x,y*) lies in (*%HS2*). This would mean that *heightimage* now contains values possibly 0 to *ch* (0 indicating ground). To represent the height of segments better, each pixel of height image is divided by the maximum possible height ch and multiplied by the maximum possible pixel value (255 for a 8-bit greyscale image). Ultimately on *heightimage* (*%HS3*), pixels would appear darker if they are part of a segment that is nearer to the ground (completely black if at height 0) and lighter as the segment is further from the ground in terms of height (completely white if at height *ch*).

Figure 4.20: Pseudocode for displaying heights of segments

```
heightimage = rows * columns matrix 0s
for x = 1 to r
      for y = 1 to c
            heightsegs(x,y) = segheight(segment(x,y))                  %HS2
      end
end
heightimage = heightimage*255/(cameraheight)                           %HS3
display(heightimage)
```

4.8 Contour matching

The fundamental matrix *F* is computed using Chen's MATLAB code [5] with the set of matching corners found earlier as input parameters. The aim is to find matching points along the contour between *image1* and *image2*.

We know that that the fundamental matrix can be used to give the equation of the epipolar line (*%CM1*) on *image2* for each point (*x,y*) on *image1*. Using this line *j = lm*i+lc*, it is possible to find a set of points within a certain region that are possible candidates for matches to (*x,y*). (Section 2.3.1) The points we are considering on *image1* are those that lie on edges, i.e. *edgeim1(x,y)*=1. Candidate matches on *image2* would be points on *image2* along the line *j=lm*i+lc* within a certain window *w* that lie on edges (*%CM2*), i.e. *image2(i,j)*=1. Since really large or small values of *lm* (gradient) can make *i* change very little or a lot respectively with respect to *j* over the window, *i* is varied over the window and *j* calculated from it for small values of *lm* (*%CM3*) whereas *j* is varied over the window and *i* calculated from it for larger values of *lm* using the equation $i = (j - lc) / lm$ (*%CM4*).

The rest of the matching process is exactly the same as that for corners. Differences between neighbouring regions are calculated and the candidate with the least difference is chosen as the matching corner (*%CM5*). After that many-one matches are removed by only keeping points on *image1* that have the least difference for a set of multiple points matching to a single point on *image2* (*%CM6*).

Figure 4.21: Pseudocode for Contour Matching

```
Given: edgeim1 (binary edge map im1)
Given: egdeim2 (binary edge map im2)

matchingpoints = ()
For each point (x,y) such that edgeim1(x,y) = 1

      lm = -{f(1,1)*x+f(2,1)*y+f(3,1)}/{f(1,2)*x+f(2,2)*y+f(3,2)};   %CM1
      lc = -{f(1,3)*x+f(2,3)*y+f(3,3)}/{f(1,2)*x+f(2,2)*y+f(3,2)};

      if lm < 20 then                                                %CM3
            for i = x-20 to x+20
                  j = lm * i + lc;
                  mintotal = 30000
                  if edgeim2(i,j) = 1 then                           %CM1
                        difftotal = sum( abs{
                              image1(x-nhood:x+nhood, y-nhood:y+nhood)
                              - image2(i-nhood:i+nhood, j-nhood:j+nhood)} )
                        if difftotal<mintotal then                   %CM5
                              mintotal = cortotal
                              tempi = i
                              tempj = j
                              found = 1
                        end
                  end
            end
      else

            for j = y-20 to y+20                                     %CM4
                  i = (j - lc) / lm
                  mintotal = 255
                  if edgeim2(i,j) = 1 then                           %CM1
                        difftotal = sum( abs{
                             image1(x-nhood:x+nhood, y-nhood:y+nhood)
                             - image2(i-nhood:i+nhood, j-nhood:j+nhood)} )
```

```
                        if difftotal<mintotal then                    %CM5
                              mintotal = diffcortotal
                              tempi = i
                              tempj = j
                              found = 1
                        end
                  end
            end
      end

      if found = 1 then
            add (x, y, tempi, tempj, difftotal) to pointtotals
      end
end

matchedpoints = ()
for each row (x, y, i, j, total) in pointtotals                %CM6
      mintotal = 65535
      for each row (a, b, p, q, itotal)
            if (p, q) = (i, j) and total < itotal
                  tempx = a
                  tempy = b
            end
      end
      add (tempx, tempy, i, j) to matchedpoints
end
```

The height for each point along the contour can be found using *height(x, y, i, j, h_matrices)* where (*x, y, i, j*) is a row of *matchingpoints* (*%CM7*). This can be later shown on image1 and it can be observed how the estimate height increases or decreases along the contour as the actual height increases or decreases. Then for each element *(x, y, i, j)* on *matchedpoints*, its height can be displayed on coordinates (*x,y*) on *image1* and (*i,j*) on *image2*, since (*x,y*) and (*i,j*) give the position of the same point on the scene on *image1* and *image2* respectively. (*%CM8*)

Figure 4.21: Pseudocode for Contour Matching (Continued)

```
display(image1)
display(image2)
for each row (x, y, i, j) in matchedpoints                     %CM6
      h <- calculateheight(x, y, i, j, h_matrices)
      display text string(h) on image1 at (x,y)                %CM7
      display text string(h) on image2 at (i,j)
end
```

4.9 Summary of implementation

This chapter has covered solutions to every sub-problem identified in Chapter 3 Design. It has also shown how the solution of each sub-problem contributes to the solution of a larger problem and how all these solutions can be put together to form the final solution to the main problem. Using the explanations and algorithms in this chapter, the entire system was coded in the Matlab language. The requirements of this system were laid out very thoroughly during Design. How well the system programmed in Matlab (using the methods described in this chapter) is evaluated in the next chapter.

5 EVALUATION

Evaluation is a vital part of any system development process. It is not possible to know whether a system meets its initial requirements without a thorough investigation of its operation using a range of experiments. A test plan is first included in this chapter explaining a range of experiments that are necessary to evaluate this system thoroughly. Afterwards, the implementation of each test and its results are discussed with a brief summary of the evaluation conducted.

5.1 The Test Plan

The aim of this project was to use several methods and techniques in order to accomplish the task of obstacle detection for a mobile robot. The three methods used are

- Obstacle detection by comparing final image to warped image
- Obstacle detection by height estimation of corners and image segmentation
- Obstacle detection by height estimation along the contours

As mentioned earlier all three methods are completely dependent on the accuracy of the computation of the homography matrix which is completely dependent on finding point correspondences on the ground plane.

The number of point correspondences would obviously vary from scene to scene and thus so would the accuracy of this obstacle detection system. Therefore the most comprehensive evaluation of the system would necessitate it being tested on a range of scenes where point correspondences are abundant to where there are hardly any. Tests were conducted on such a range of scenes and the relation between the accuracy of obstacle detection and scene features (presence of corners and point correspondences) are observed.

Furthermore other factors for example the presence of texture on objects and the floor can affect obstacle detection a great deal. The absence of texture on the object would mean that when comparing the warped and final images, only the contour of the object (rather than the whole thing) show up as a disparity indicating an obstacle. However, if there is too much texture on the ground, because of the imperfection of the homography matrix, detailed textures will not exactly match between the final and warped images and may be consequently detected as obstacles. These experiments were also conducted on a range of objects with different amounts of texture on them.

Objects come in varying sizes and shapes. Some methods such as corner height estimation would obviously not work on spherical objects at all. Several tests were conducted with objects of varying shapes to test the range of functionality of the obstacle detection system.

Other factors such as object height relative to camera height, distance of obstacles from camera and so on also contribute to how satisfactory the results obtained from the system and their influences were examined thoroughly with several experiments conducted.

Please note that since the RANSAC algorithm does not always compute the same value of H or F for a given scene, some test results were simply unpredictable and inaccurate over different runs. Only the best results are shown in the figures. All experiments were conducted with the camera at 75cm from the ground and the camera pointed 30 degrees down from the horizontal.

5.2 Point Correspondences

5.3.1 Corner detection

As mentioned earlier this is initially dependent on how well corners can be found on both image and then a correspondence relationship can be established between pairs of corners each of which are on a different image.

The Harris corner detector (coded by Peter Kovesi [15]) was used to perform this initial task of locating corners. The results were very encouraging since in most occasions a large number of corners could be found. Moreover, even when the ground plane and the object were very similar in colour the Harris detector was able to locate the corners which is considered quite difficult.

5.3.2 Matching correspondences

Matches of pairs of corners were found by initially taking one pixel on *image1* and then looking at the neighbourhood of every pixel on *image2* within a certain radius of it and picking the closest matches.

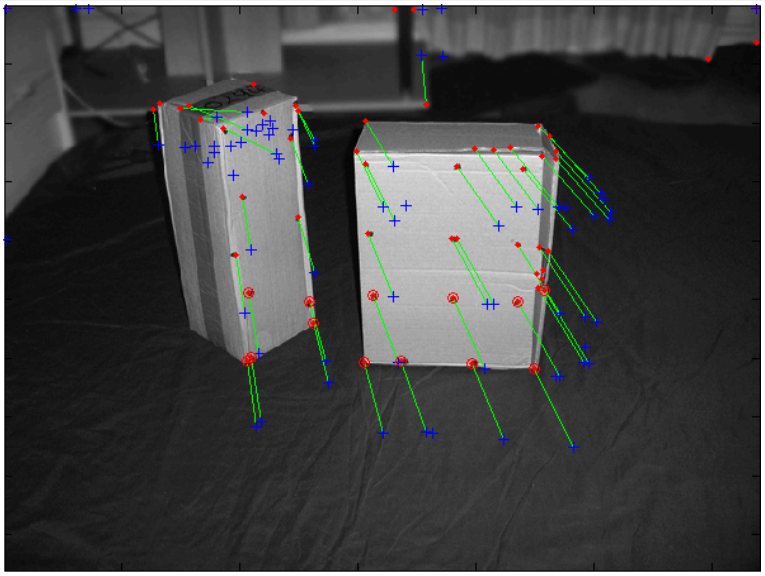
As expected if there were too many corners in the neighbouring regions, the process was considerably slow and took up to three minutes for an image with around a 200 corners (Figure 5.1b).

The other problem faced was when there were too many putative matches that had similar neighbourhoods as the corner being matched. This is very likely in floors which have a chessboard-type pattern with very small blocks (Figure 5.1d). Let us assume, a corner on the top right of a black block on *image1* is being matched. If the pattern is really dense then it is possible there might be several top right corners on black blocks on *image2* within the window of search. This means that several candidate matches may have exactly the same neighbouring regions as the initial corner and it may be matched with any one of these candidates not necessarily the right one, eventually leading to degeneracies in the computation of the homography of the scene.

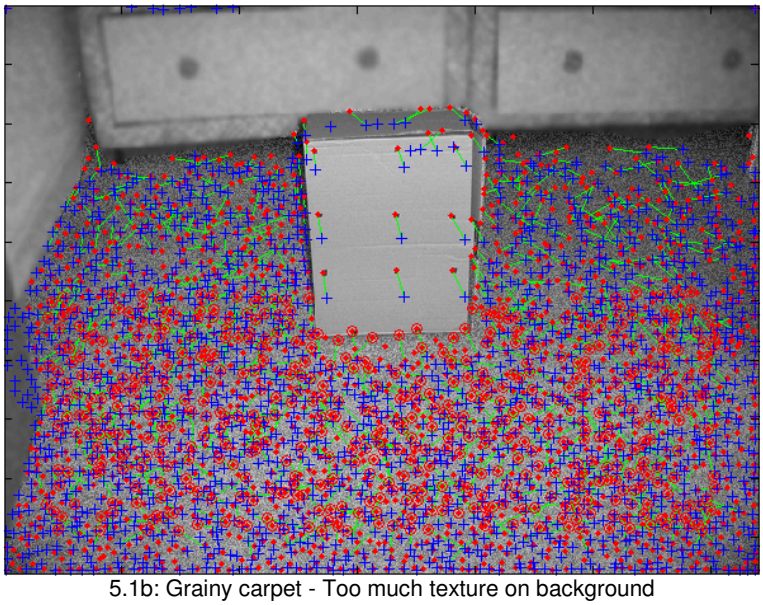
If there is too much texture however, e.g. a grainy pattern on the floor, the corner detector may detect literally thousands of corners on the scene and may take a while processing matches which will be highly inaccurate (Figure 5.1b). However if the pattern is not very distinct, it does not regard any of the grains as corners at all and so no corners or point correspondences can be found.

A mid-way scene in terms of finding good point correspondences is one with a few corners on the ground e.g. pieces of paper on the ground (Figure 5.1c). Ground planes such as this are used through out the latter stages of evaluation. The best corner correspondences can probably be found with a tiled floor where each tile (unlike the chessboard) is big enough for there to not be two similar corners on the search window of each corner being matched (Figure 5.1e).

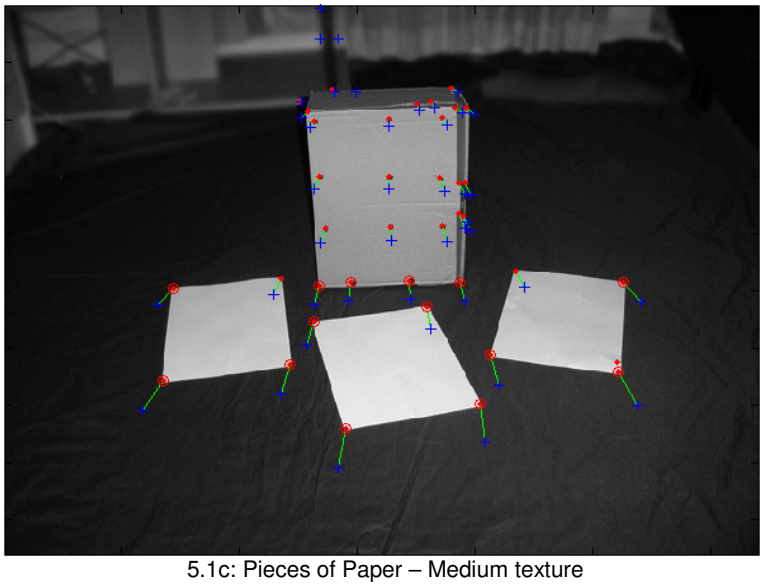
Please Note: The figures show corners on the initial image as dots and the corners on the final image as crosses. There is a line between a dot and a cross if the pair is matched corner. Corners with circles around them are those considered reliable for homography computation.



5.1a: No texture at all on background

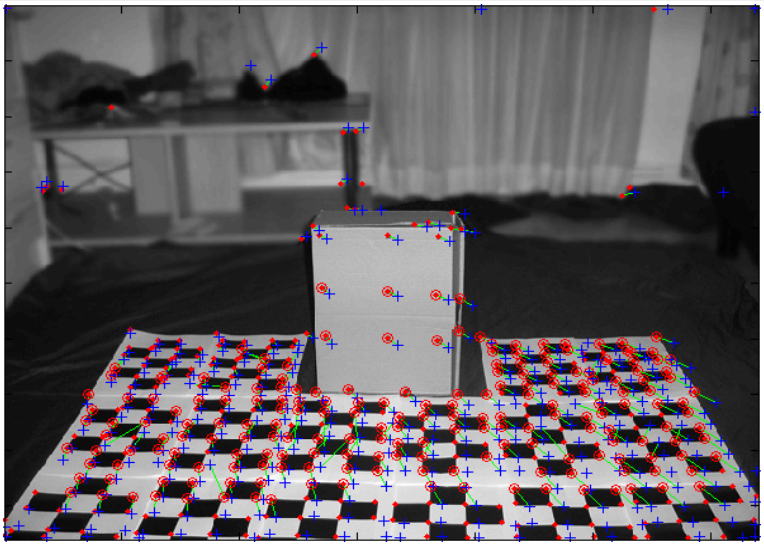


5.1b: Grainy carpet - Too much texture on background

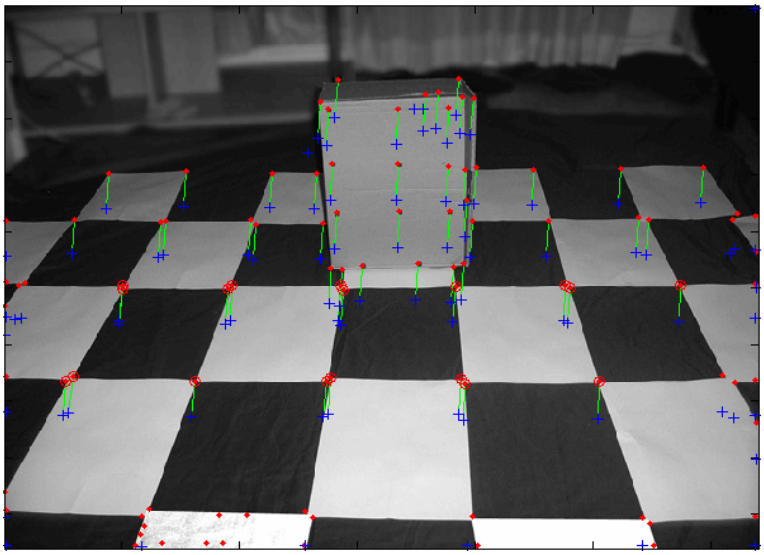


5.1c: Pieces of Paper – Medium texture

Figure 5.1: Scenes with different ground planes



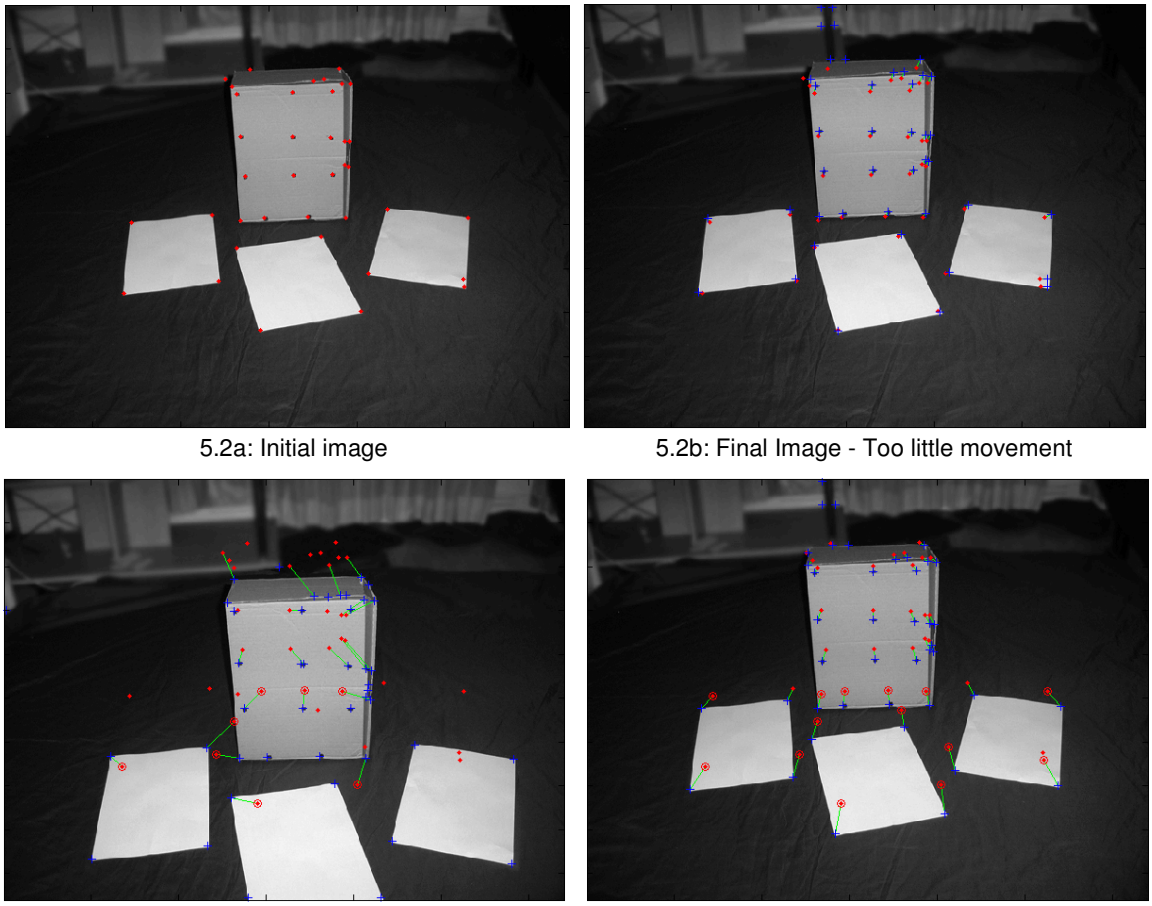
5.1d: Frequent tiles – Too much texture



5.1e: Large Tiles – Most desirable texture

Figure 5.1: Scenes with different ground planes (continued)

Point correspondences are constrained by the window used to search for candidate matches. This means that if the camera is moved forward or backward too much between the two images, the actual match for an initial corner may not be within its search window and will not be considered as a candidate match for it (Figure 5.2c). This means the corner is not matched to anything at all or in the worse case inaccurately to a random candidate corner within its window. On the other hand if the movement is too small (Figure 5.2b), most point matches will be considered as too close and thus unreliable for computing H and they will be disregarded. Camera movements between 5cm and 10cm were considered the most suitable for these set of experiments.



5.2a: Initial image

5.2b: Final Image - Too little movement

5.2c: Final Image - Too much movement

5.2d: Final Image - Adequate movement

Figure 5.2: Movement of camera between images and its effects

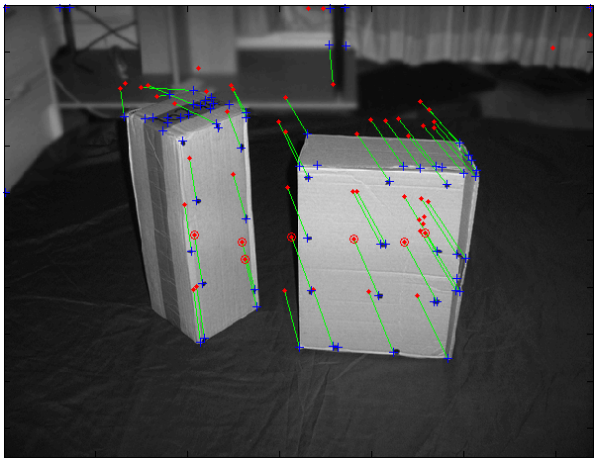
5.3 Computing homography and corner heights

As mentioned earlier the computation of H is entirely dependent on the location of point correspondences between the two images of the scene. Estimation of heights of corners and points along contours are done at the same time and so these were evaluated at the same time as the computation of H.

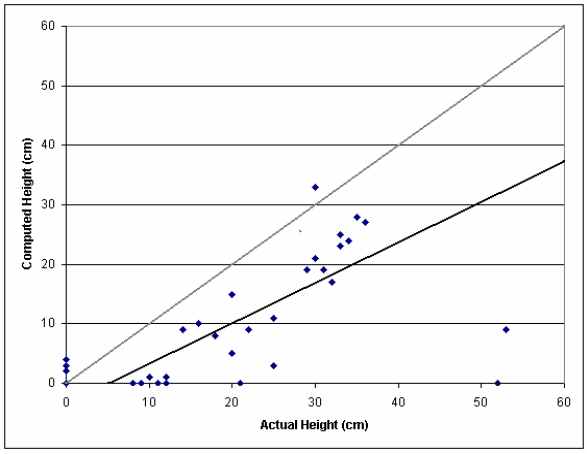
How well H has been computed can be observed by having a look at both images with each initial corner, its matched final corner and its warped position estimated from the computed value of H. The closer the final and warped positions of each corner on the ground are, it can be assumed the more accurate the computation of H was. Moreover if the computation of H is accurate, then the computed heights of corners on the ground and obstacles would be consistent with their real heights. Graphs of actual heights and computed heights along with the correlation coefficients between them give a good picture of how well the homography for the scene has been computed.

Corners in the background did not have much on an effect on the computation of H, since only corners on the lower half on the image were taken into account while performing the RANSAC algorithm.

Inaccurate point correspondences on the floor owing to too much texture on it may lead to degenerate computations of H. As a result heights estimated will not be accurate either.



5.3a: Final Image with corners



5.3b: Computed vs Actual Heights
(Black = regression line, Grey = perfect computation)

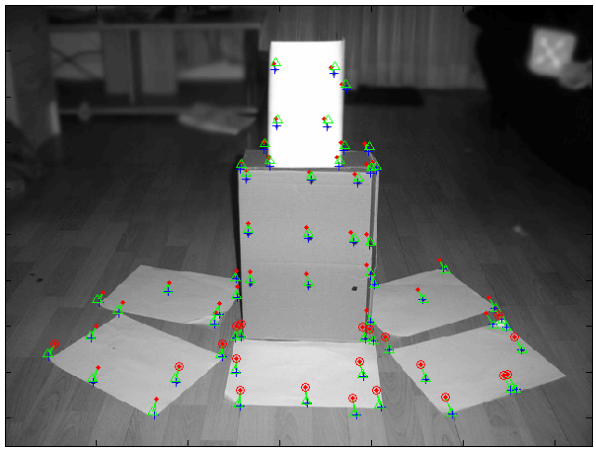
5.3c: Final Image - Heights of Corners

Figure 5.3: Degenerate estimation of H and the resulting height measurements when a plane other than the ground is selection for homography computation in the absence of enough corners on the ground.

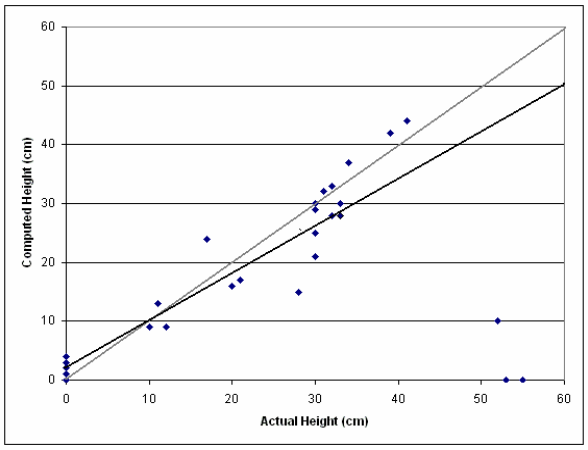
If there are no point or few correspondences on the ground (that is there is either too much texture on it or none at all), the RANSAC algorithm attempts to compute H by using the corners on the base of the obstacle. This does not lead to a very accurate estimation of H since there are very few such corners usually present. Moreover this is only possible if the base of the object lies in the lower half of the image since anything above it are disregarded during the computation of H. It is also possible for RANSAC to chose points that are on a different plane above the ground to compute H. E.g. in Figure 5.3a, corners on the 10cm plane (marked with circles) are chosen by RANSAC for the final computation of H. As a result, all computed heights of corners are about 10cm less than their actual heights with 10cm being identified as 0cm (Figure 5.3c). There is a still strong correlation between actual and computed heights as seen in Figure 5.3b, with the correlation coefficient computed as 0.75,

however there is an x-intercept of around 6cm (which is almost equal to the height of the plane H corresponds to)

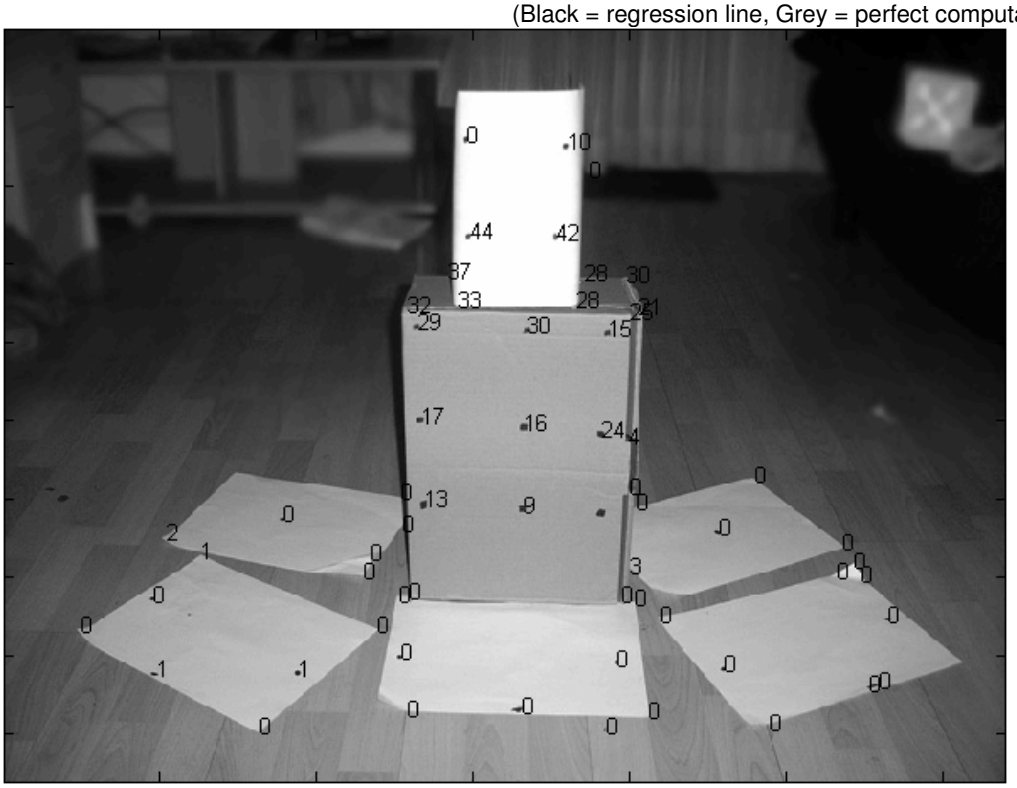
As mentioned earlier, with a few flat objects (e.g. pieces of paper) on the floor, a decent estimation of H can be expected (Figure 5.4a). Corners marked with circles are used for the computation of H and all these lie on the ground plane. As one can see the regression line of computed vs actual heights is very close to the line of perfect computation (where computed height = actual height). The correlation coefficient was computed at 0.80 which is quite decent. Moreover the line almost passes through the origin. A floor with large tiles would yield the maximum point correspondences and thus the best computation of H. Another point to note is that height estimations are highly inaccurate for heights close to the camera height. This can be observed in Figure 5.4b where the camera is at 75cm.



5.4a: Final Image with corners



5.4b: Computed vs Actual Heights
(Black = regression line, Grey = perfect computation)



5.4c: Final Image - Heights of Corners

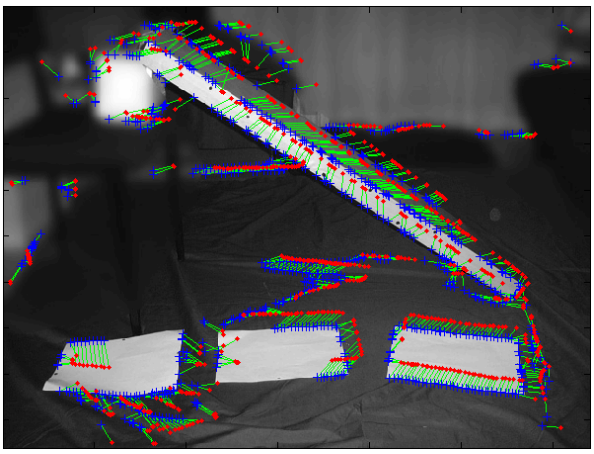
Figure 5.4: Decent estimation of H and the resulting height measurements with sufficient point correspondences on the ground plane. The correlation coefficient computed between actual and computed heights indicates the accuracy of the computation

5.4 Computing epipolar geometry and contour heights

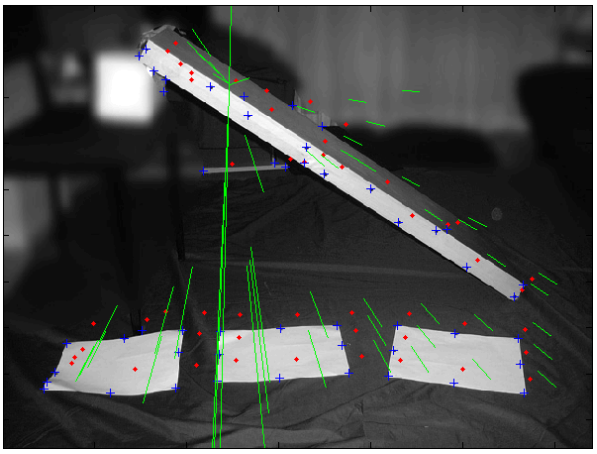
As with the computation of the homography of the scene, the computation of the fundamental matrix is completely dependent on the location of point correspondences between the initial and final images.

The accuracy of the estimation of F can be observed by having a look at the epipolar lines of each initial corner and checking whether the matching final corners lie on the corresponding epipolar lines. Furthermore, point correspondences along the contours can be found using the fundamental matrix to define a search trip along the epipolar lines for each point on the contours. How well these points match keeping in mind the camera motion is an even better indicator of the how good the computed estimate of F is. As in the case of the homography, inaccurate point correspondences lead to very degenerate computations of epipolar geometry. For scenes with few point correspondences the fundamental matrix computed was not very accurate and lead to many inaccurate point matches along the contours. For scenes with a decent number of correspondences the fundamental matrix computed was quite a good estimate and the point matches along the contours were more or less perfect. For scenes with an abundant number of correspondences the computation of F was more or less perfect and so were the consequent point matches along the contours.

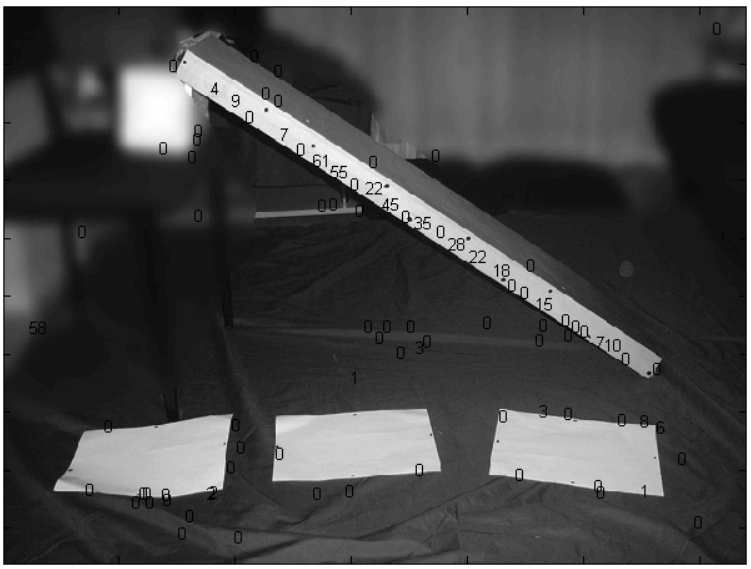
Figure 5.5 shows a decent computation of epipolar geometry owing to good point correspondences on the ground and well-performed edge detection. Epipolar lines and edge heights are shown as well. Finally in 5.5d, the obstacle detection methods using heights of edges (in blocks) are shown, with lighter blocks indicating greater heights.



5.5a: Final Image - Contour matching

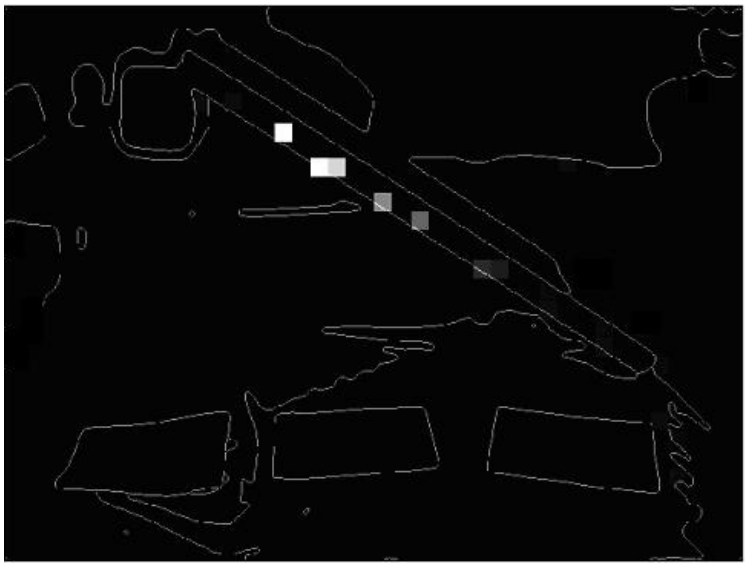


5.5b: Final Image - epipolar lines of corners



5.5c: Final Image - Contour Heights

Figure 5.5: Contour matching, epipolar lines and heights along contours



5.5d: Obstacle detection via Contour Heights

Figure 5.5: Contour matching, epipolar lines and heights along contours (Continued)

5.5 Image warping and obstacle detection

Since the pixels of the warped image can be thought of as functions of pixels of the initial image with the homography matrix as a parameter, any obstacle detection method using the warping image would be dependent on the accuracy of the homography matrix. We can simply look at the final image, warped image, the difference image and the block difference image to observe how well the system has performed at detecting obstacles.

As expected degenerate computations of the homography matrix (e.g. where corners on a plane other than the ground have been used to compute it because of the lack of corners on the ground or simply where point correspondences were computed incorrectly) objects on the ground may easily be detected as obstacles.

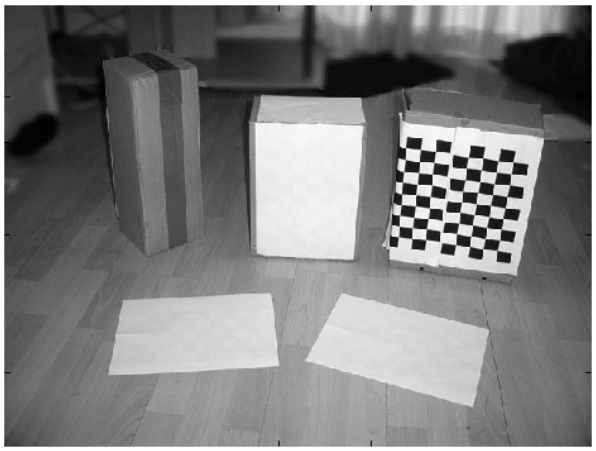
If there are too few corners on the ground, the estimate for H is not very accurate and thus the warped image will not be a very good representation of the initial image after the camera motion. This may lead to objects on the floor being detected as obstacles as well.

In the case of objects with very little texture on the ground (assuming the homography computed is a decent estimation), only blocks around the edges of objects will be detected as obstacles. This is simply because when comparing the final and warped images, there is not much difference towards the centre of the object because of the lack of texture. However the pixels are obviously conspicuously different along the contours of the object.

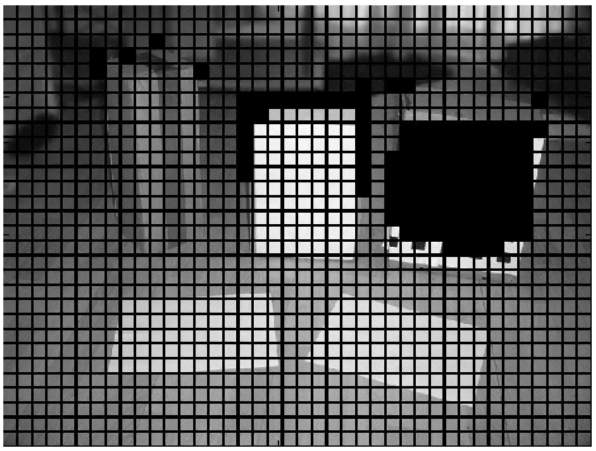
Even though this performs the task of obstacle detection to some extent, it is quite fallible. For example, in the case where the object and the ground behind it are of more or less the same colour (brown box on Figure 5.6b), there will not be much difference in pixels values even along the contours and thus no obstacles will be detected at all. One must also realise that depending on the threshold set for block differences, this method can only detect obstacles over a certain height (around 10cm in these experiments). Setting the threshold too high means that most objects will not be detected as obstacles and setting it too low means that even the slightest texture on the ground may be identified as obstacles.

Even if a large number of good point correspondences are used to compute H, it is still merely an estimate. For this reason, if there is a lot of texture on the ground in the initial image, they will not be transferred perfectly to the warped image and this might lead to inconsistencies with the final image. As a result, anything with texture on the ground may be detected as an obstacle (Figure 5.6d).

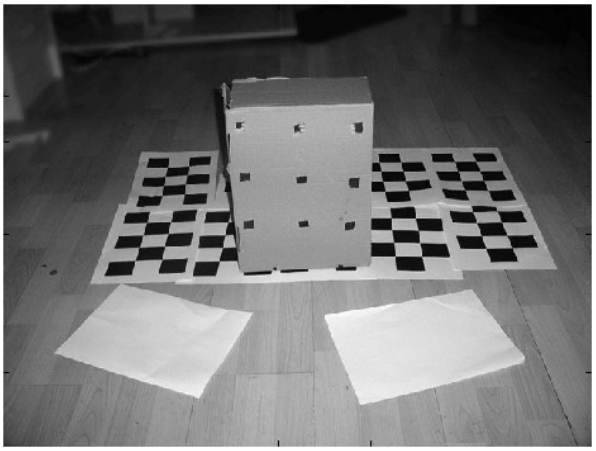
The best case scenario for this method of obstacle detection is perhaps when the computation of H is decent, there is very little texture on the ground and a lot of it on the obstacles (square patterned box on figure 5.6b).



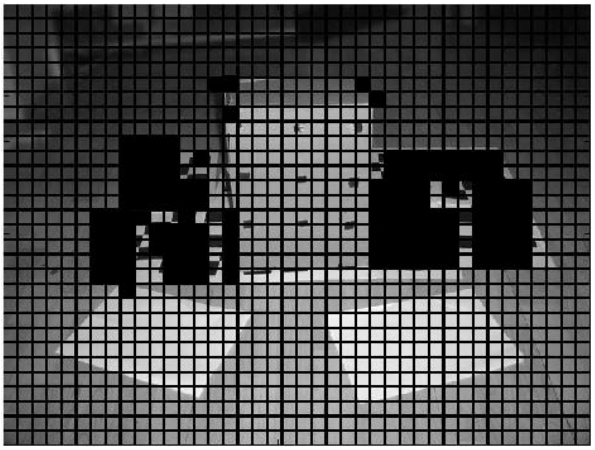
5.6a: Objects with no texture and a lot of texture



5.6b: Obstacle detection via image warping



5.6c: Ground with a considerable amount of texture



5.6d: Obstacle detection via image warping

Figure 5.6: Image Warping with different objects and backgrounds
Scenes with objects with some texture and the ground with some texture in parts, demonstrating how too much texture on the ground may be detected as obstacles and how objects with little texture on them may not be well recognised as obstacles. It also shows how obstacles similar in colour to the ground may not be detected at all (e.g. brown box In 5.6a)

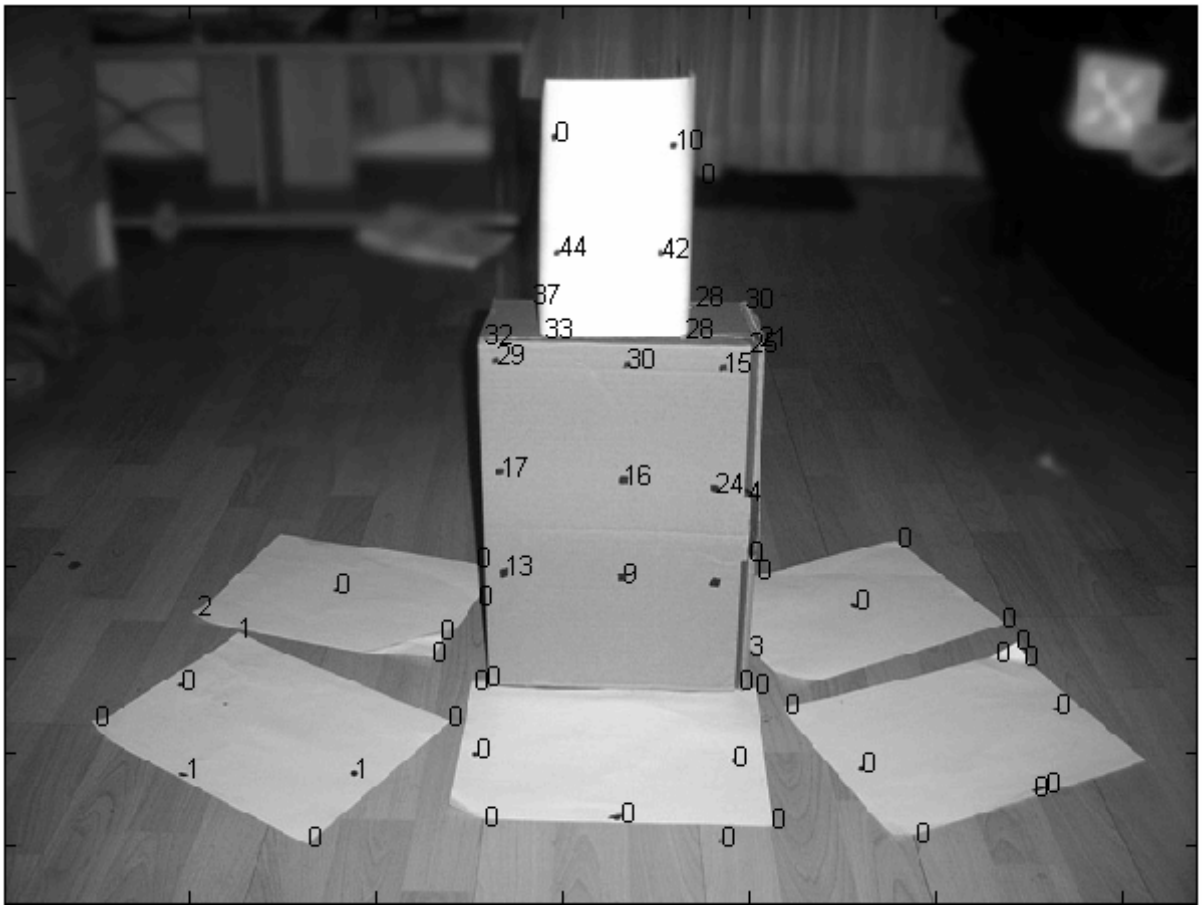
The image warping method seems to work for all shapes of objects given there is enough texture on the object and the homography is computed more or less accurately.

5.6 Segmentation and obstacle detection

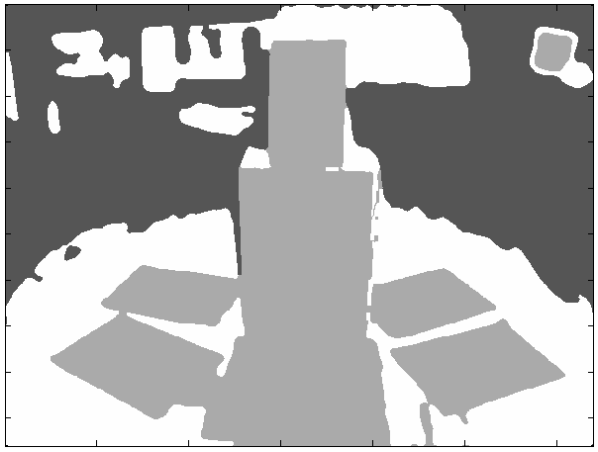
This depends on how well H has been computed (consequently how well the heights of corners have been estimate) and how easily the image can be segmented.

Assuming a decent computation of H, there are mainly two cases we have to consider.

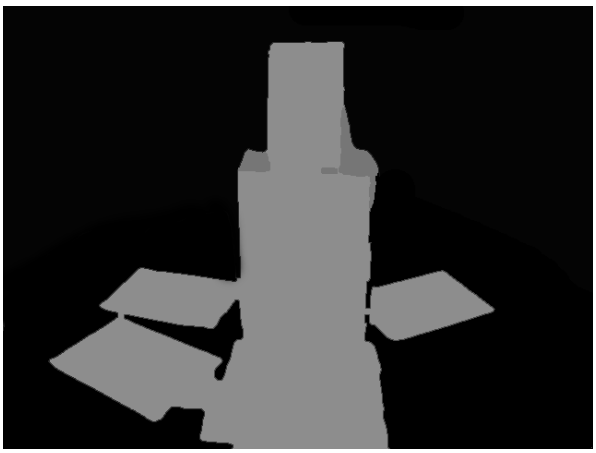
If the obstacle and ground are more or less the same colour, they may be segmented as one object (Figure 5.7b). So the maximum height of a corner in the object is considered the height for all points on the segmented object. As a result all points on the ground would be assigned the same height and the entire ground would be indicated as an obstacle (Figure 5.7c).



5.7a: Final Image - Heights of Corners



5.7b: Final Image – Segmentation

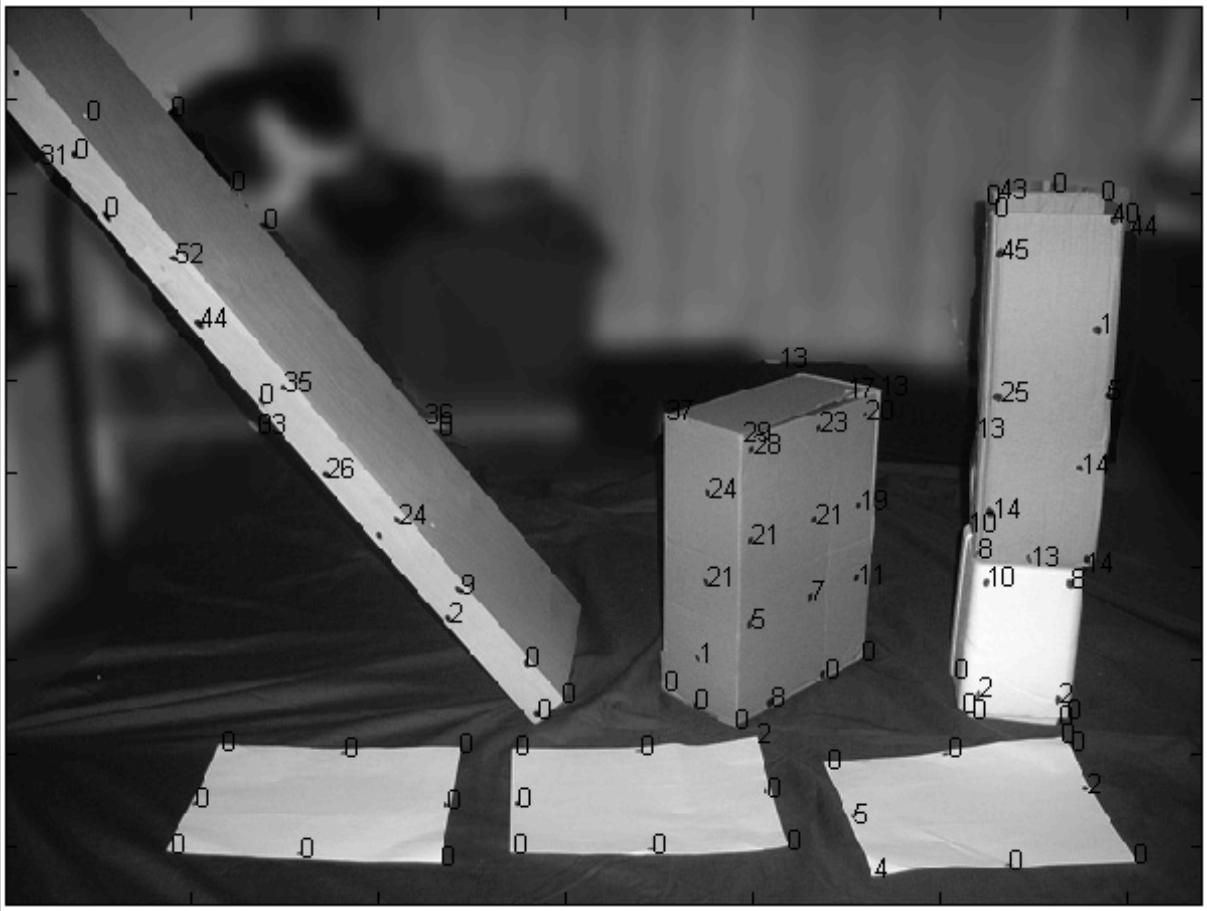


5.7c: Final Image – Obstacle detection

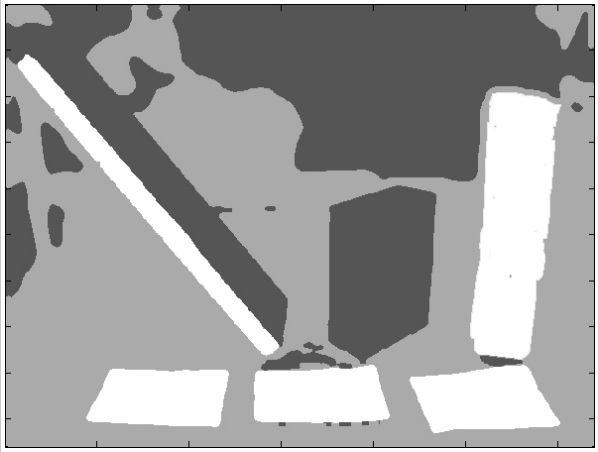
Figure 5.7: Segmentation and Obstacle Detection - Obstacle and background similar in colour

If the obstacles and the ground are different colours, they are segmented as separate objects. As a result pixels on the ground are given more or less the height of 0 (depending on how good the

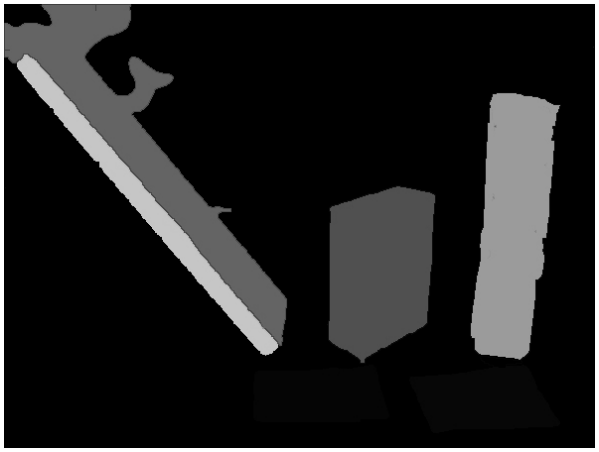
estimation of H is). Obstacles on the other hand are treated individually and are given more or less accurate heights (Figure 5.8).



5.8a: Final Image - Heights of Corners



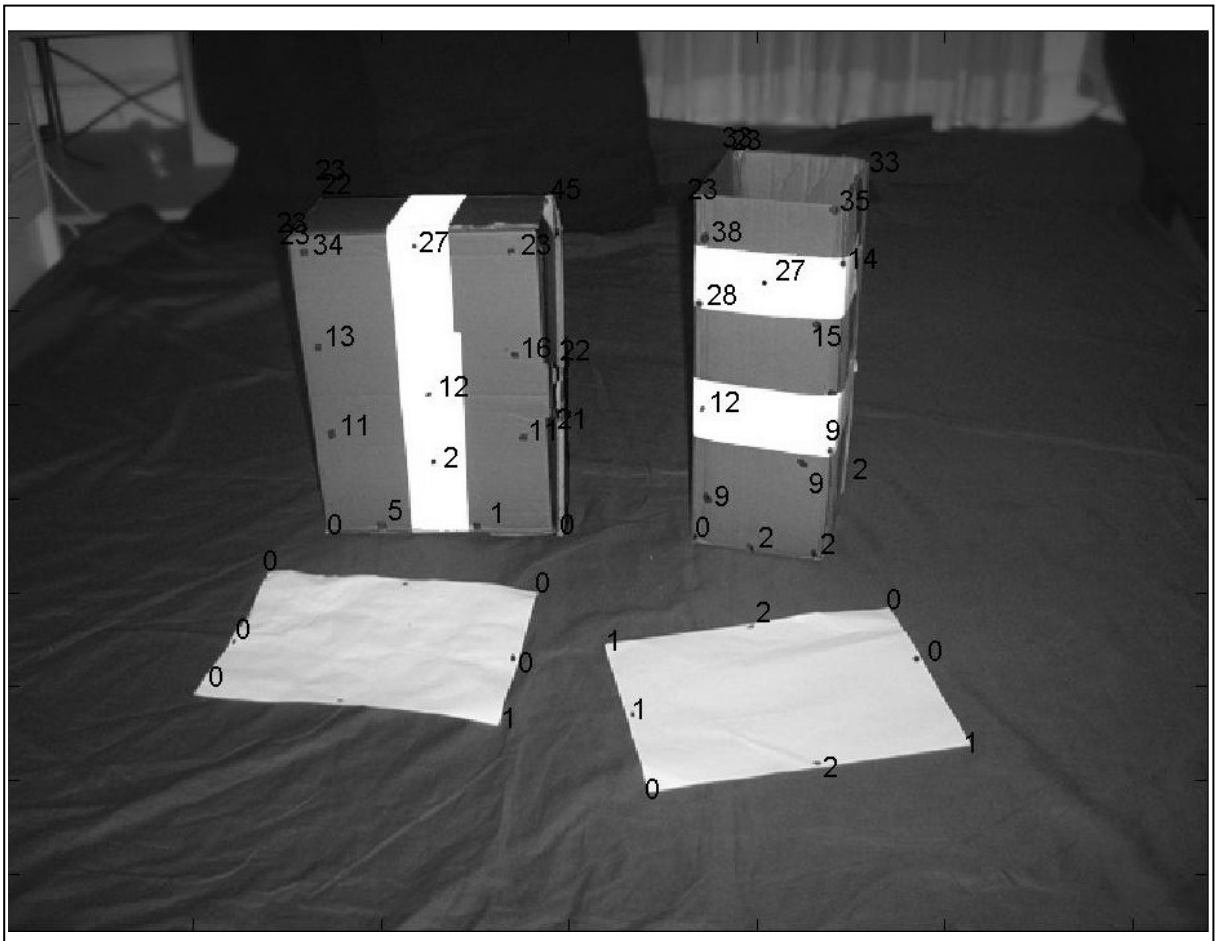
5.8b: Final Image - Segmentation



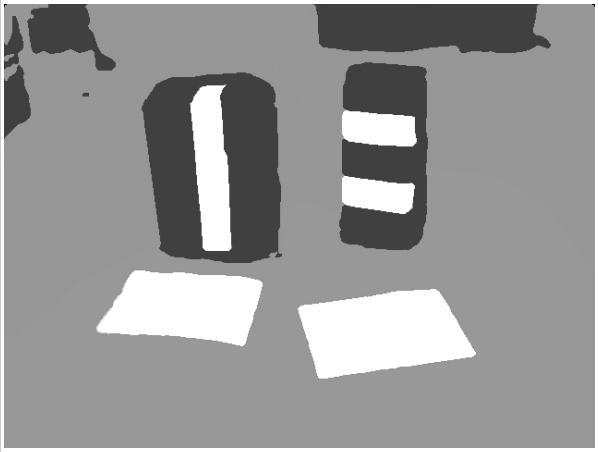
5.8c: Final Image - Obstacle detection

Figure 5.8: Segmentation and Obstacle Detection - Obstacle and background different in colour

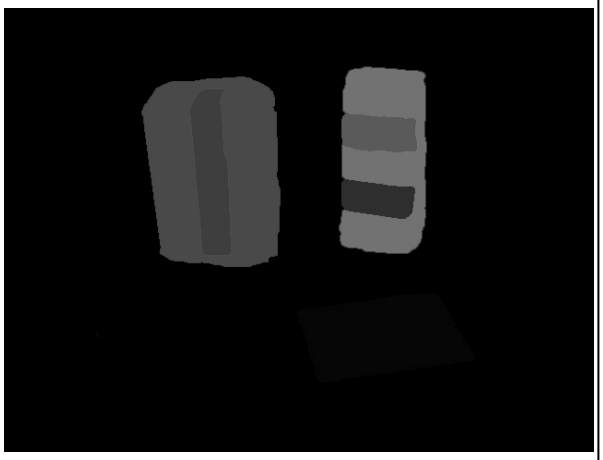
However, if the object has quite a bit of texture in it (e.g. stripes), it may be segmented as several objects and each segment may have its own height (Figure 5.9). e.g. If the object has horizontal stripes, it will be divided into horizontal segments with each segment having a progressively higher height value (Figure 5.9).



5.9a Final Image - Heights of Corners



5.9b: Final Image – Segmentation



5.9c: Final Image – Obstacle detection

Figure 5.9: Segmentation and Obstacle Detection - Stripes on obstacle

5.7 Summary of Evaluation

A range of experiments have been conducted to evaluation the performance of the solution to each basic sub-problem (e.g. corner detection) to the three obstacle detection methods used in this project. From the experiments conducted it is obvious that each of the three methods of obstacle detection do not perform their best under the same set of scene characteristics. The following is a table of methods of obstacle detection and the features of an ideal scene for each of them.

Method of Detection / Preferred Scene Features	Comparison of Warped to Final Image	Segmentation and heights at corners	Height computation along contours
Number of point correspondences	High	High	High
Distance covered by camera motion	Medium	Medium	Medium
Frequency of repeating patterns on ground	Medium	Medium	Medium
Texture on ground	Low	Low	Low
Texture on object	High	Low	Any
Height of obstacle	High	High	High
Relative colours of obstacle and ground	Dissimilar	Dissimilar	Dissimilar
Shape of obstacle	Any	Not spherical	Slanting

Table 5.1: Methods of obstacle detection and preferred scene features

6 CONCLUSION

The concluding chapter of this report is a brief summary of each of the previous chapters. It also gives a summary of evaluation of the research conducted in this project. It finally goes on to suggest possible future work emphasising on the drawbacks of the system developed.

6.1 Summary of Report

Chapter 1 gave an introduction to obstacle detection and its applications in robotics. It also introduced a brief outline of the aims of this project.

In Chapter 2 "Literature Review", all the background information required in computer vision, image processing and pattern recognition required to implement this project were discussed in great detail. A brief history of computer vision was given following its definition. Concepts fundamental to this project such as multi-view relation, planar homography, epipolar geometry and segmentation were discussed in great detail. Methods for computing the homography and epipolar geometry of a scene were also discussed. This also necessitated the discussion of topics such as corner and edge detection which are essential in these computations.

Chapter 3 entitled "Design" started off with a more comprehensive list of requirements and introduced the three methods of obstacle detection researched in this project. It broke down the main problem into a series of progressively smpler sub-problems. Each sub-problem was described briefly with references to Chapter 2 Literature Review indicating possible solutions where appropriate. A list of the hardware constraints on the system were provided. Several design methodologies and development methods were also discussed before selecting the most appropriate one.

In Chapter 4 "Implementation", solutions to each sub-problem were described in great detail with reference to the chosen development environment. Diagrams and pseudocode were provided along with references to the literature review to help aid the descriptions.

Chapter 5 "Evaluation" begins with a test plan created to evaluate the functionality of the system implemented from Chapter 4. A wide range of tests are conducted on each of the main sub-problems under very difference circumstances or scenes. The results are then summarised indicating the reliability of the obstacle detection system as a whole.

6.2 Research Results

Chapter 5 entitled "Evaluation" discussed in great detail the outcome of the research conducted on this project. Each of the three obstacle methods used were put to the test under a range of different scenes, with different objects and different backgrounds to test their reliability as fully as possible.

Each method seems to have its limitations despite how theoretically sound they may be. For example, the image warping method seems to only find detect obstacles above a certain height depending on a threshold set. If the threshold is too high, then it detects texture on the ground as obstacles as well and if it is too low, objects of decent heights are not detected at all. It also counts on there being too much texture on the object and very little on the ground which is not always true for a real scene.

The epipolar geometry method using contour heights seems to be very theoretically sound as well however depending on the computation of the fundamental matrix (which is dependent on simply too many scene features) the epipolar lines were not always very accurate. This led to inaccurate matches

for edge points between images and thus inaccurate heights of points. However the method was still quite reliable in most scenes unless there was much noise in the images.

The image segmentation method is perhaps the most sound in theory and is the most fallible in practice. It seems to count on there being minimum texture on objects and grounds and the presence of very distinct corner points on both images. Broken edges due to noise or similarity in colour between obstacles and ground planes can often lead to very poor obstacle detection. A good example would be when because of similar colours the obstacle and ground plane is segmented as one object and all of it is indicated as an obstacle.

Other than that the results were simply unpredictable at times. This may simply be because the RANSAC algorithms used to compute the homography and fundamental matrices do not always yield the same results. More importantly the entire system can be considered a pipeline process with corner detection and matching being the start of the pipeline. If decent point correspondences are not found on the ground the entire system simply breaks down. Degenerate computations of H and F have led to some highly inaccurate and often hilarious results in the absence of distinct points on the ground.

As sound as the theories of computer vision are, they have not yielded the best results under a wide range of scenes. However, given certain restrictions on the scene (Section 5.7) such as the presence of distinct corners on the ground, sufficient camera movement, minimum texture on ground an so on the results have proven the theories very sound. Under these conditions, obstacle detection using computer vision can be considered quite reliable.

6.3 Further Work

The methods of obstacle detection were decent enough for an undergraduate project but not really reliable enough for the navigation of an autonomous vehicle. This section gives a few suggestions on how the reliability of methods of obstacle detection may be improved.

Perhaps images with better resolution could be used to get better estimates of homography and epipolar geometry of scenes. This could however lead to the presence of more unreliable corners and broken edge lines which may make matters worse during the homography computation and image segmentation respectively.

Several optimisations could be made to enhance the reliability of the methods. For example, if an obstacle and the ground get segmented together, epipolar geometry and contour height estimates could be used to detect where the ground ends and where the object starts. A horizontal line can be drawn separating the obstacle and the ground marking them with their appropriate heights.

At the moment, the obstacle detection system is simply a digital camera on a tripod being moved around manually between images for experiments. The same experiments could be conducted by an actual mobile robot with either an on board computer system or a connection to a personal computer to test the reliability of obstacle detection using computer vision in real time.

BIBLIOGRAPHY

References

[1] H.G. Barrow and J.M. Tenenbaum: “Recovering intrinsic scene characteristics from images Computer Vision Systems”, New York Academic Press, 1978.

[2] D. Blodgett: “Discussion of Edge Detection Methods”, Rochester Institute of Technology 2004.

[3] C.R. Brice and C.L. Fennema: “Scene Analysis Using Regions", Artificial Intelligence 1:205-226, 1970.

[4] J. Canny: “A Computational Approach to Edge Detection”, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1986.

[5] Z. Chen: “K-means clustering” and “RANSAC normalised 8 point compuatation of F” MATLAB codes, Computer Vision Research, University of York, Department of Computer Science, 2004.

[6] A.G. Bors, E.R. Hancock and R.C. Wilson: “Computer Vision” lecture notes, University of York, Department of Computer Science, 2004.

[7] M.J. Daily: “Color Image Segmentation. In Advances in Image Analysis”, R. Gonzalez and Y. Mahdavieh, eds., SPIE Press, 1993.

[8] O. Faugeras and Q. Luong: “The Geometry of Multiple Images”, The MIT Press, 2001.

[9] M.A. Fischler and R.C.Bolles: “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”, Comm. of the ACM, Vol 24, 1981.

[10] A. Guzman: "Computer Recognition of three-dimensional objects in a visual scene", Tech. Rep. MAC-TR-59, AI Laboratory, MIT, 1968.

[11] C.J. Harris and M. Stephens: “A combined corner and edge detector”, 4th Alvey Vision Conf., Manchester, 1988.

[12] R. Hartley and A. Zisserman: “Multiple View Geometry in Computer Vision”, Cambridge University Press, 2000.

[13] B.K.P. Horn: "Obtaining Shape from Shading, The Psychology of Computer Vision”, McGraw Hill, 1970.

[14] R. Kasturi and R.C. Jain: “Computer Vision Principles”, IEEE Computer Society Press, 1991.

[15] P. Kovesi: “Harris Corner Detector” MATLAB code, Department of Computer Science & Software Engineering, The University of Western Australia, 2002.

[16] L. Zollei: “Place Recognition Using Colour Region Analysis”, Mount Holyoke College, 1999.

[17] P.R.S. Mendonça: “Affine Epipolar Geometry from Profiles”, Computer Vision and Robotics Group, University of Cambridge, 2000.

[18] R.C. Wilson: “Pattern Recognition” lectures notes, University of York, Department of Computer Science, 2004.

[19] L. Roberts: "Machine perception of three-dimensional solids" “Optical and electrooptical information processing”, Masachussets, MIT Press, 1965.

[20] S. Singh and P. Keller: “Obstacle Detection for High Speed Autonomous Navigation”, IEEE Proceedings of International Conference on Robotics and Automation, 1992.

[21] H. Spies: “Parameter Estimation and Calibration”, Computer Vision Laboratory, Department of Electrical Engineering, Linköping University, 2003.

[22] S. Teller: “Image Segmentation”, Masachussetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, 1996.

[23] J.M. Tenenbaum and H.G. Barrow: "A paradigm for integrating image segmentation and interpretations.", International Joint Conference on Pattern Recognition 3, 1976.

[24] T.A. Wilkinson: "A High-Performance Vision System for Obstacle Detection", The Robotics Institute, Carnegie Mellon University, 1997.

[25] K.H. Wong: “3D Computer Vision: Feature Extraction”, Department of Computer Science and Engineering, The Chinese University of Hong Kong, 2004.

APPENDIX A: CODE

```
% Main program

file1 = 'image1.jpg'                % Initial input image file
file2 = 'image2.jpg'                % Final input image file

% Choosing which methods of obstacle detection to use
segmenter=1
edger=1
warper=1

image1=imread(file1);
[row,col,color]=size(image1);
image2=imread(file2);


% Corner Detection
sigma=2;
thresh=1000;
radius=2;
disp=1;
[iscornerim1, r1, c1] = harris(image1, sigma, thresh, radius, disp);
[iscornerim2, r2, c2] = harris(image2, sigma, thresh, radius, disp);

% Image segmentation and edge detection
[im edgeim1] = kmean_seg(file1);
[im edgeim2] = kmean_seg(file2);



%%%%%%%%%%%%%%%%%%%%
% Matching Corners %
%%%%%%%%%%%%%%%%%%%%

cornertotals = []
% Matching Corners resulting in one-many pairs

for z=1:size(r1)
    x = r1(z);
    y = c1(z);

    if x>windowsize & y>windowsize & x<row-windowsize & y<col-windowsize

        mintotal = 30000;

        found = 0;

        for i=(x-windowsize):(x+windowsize)

            for j=(y-windowsize):(y+windowsize)

                if i>nhood & j>nhood & i<row-nhood & j<col-nhood &
iscornerim2(i,j)==1

                    diff_total = sum(sum(abs(double(image1(x-
nhood:x+nhood,y-nhood:y+nhood)) - double(image2(i-nhood:i+nhood,j-
nhood:j+nhood)))));
```

```
                    if diff_total<mintotal
                        mintotal=diff_total;
                        tempi = i;
                        tempj = j;
                        found = 1;
                     end

                end

            end

        end

        if found>0

            cornertotals = [cornertotals; x y tempi tempj mintotal];

        end

    end

end

[nomatchedcorners tmp] = size(cornertotals);
matchedcorners = [];


% Matching Corners resulting in one-one pairs

tempx=0;
tempy=0;

for u=1:nomatchedcorners

    i = cornertotals(u,3);
    j = cornertotals(u,4);
    mintotal = 65535;

    for v=1:nomatchedcorners

        a = cornertotals(v,1);
        b = cornertotals(v,2);
        p = cornertotals(v,3);
        q = cornertotals(v,4);
        dtotal = cornertotals(v,5);


        if  i==p & j==q & dtotal < mintotal

            tempx = a;
            tempy = b;
            mintotal = dtotal;

        end

    end

    matchedcorners = [matchedcorners; tempx tempy i j mintotal];

end
```

```
[nomatchedcorners tmp] = size(matchedcorners);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Homography Computation Using RANSAC %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



dis = [];
h_corners = [];

% Discarding unreliable and non-ground corners

for i = 1:nomatchedcorners

    distance = ((matchedcorners(i,1)) - (matchedcorners(i, 3)))^2 +
((matchedcorners(i,2)) - (matchedcorners(i, 4)))^2;

    if distance > 25 & matchedcorners(i,1)>row/2
        h_corners = [h_corners; matchedcorners(i, :)];
    end

end


% Displaying all corner correspondences indicating those used for H
Computation

figure, imagesc(image1), axis image, colormap(gray), hold on
title('Initial Image - Point Correspondences');

for i=1:nomatchedcorners
    plot(matchedcorners(i, [2 4]),matchedcorners(i, [1 3]), 'g-');
end

plot(c1, r1, 'r.')
plot(c2, r2, 'b+'),
plot(h_corners(:,2), h_corners(:,1), 'ro');


figure, imagesc(image2), axis image, colormap(gray), hold on
title('Final Image - Point Correspondences');

for i=1:nomatchedcorners
    plot(matchedcorners(i, [2 4]), matchedcorners(i, [1 3]), 'g-');
end

plot(c1, r1, 'r.')
plot(c2, r2, 'b+'),
plot(h_corners(:,2), h_corners(:,1), 'ro');




[nohcorners tmp] = size(h_corners);
mintotald = 655365

% Starting RANSAC (100 times)
```

```
for q = 1:100

    t1 = 0;
    t2 = 0;
    t3 = 0;
    t4 = 0;
    x=0;

    while (t1==t2|t1==t3|t1==t4|t2==t3|t2==t4|t3==t4)
        x=x+1;
        randoms = int16(rand(1,4)*(nohcorners-1)+1);
        t1 = randoms(1);
        t2 = randoms(2);
        t3 = randoms(3);
        t4 = randoms(4);
    end


    x1 = h_corners(t1, 1);
    y1 = h_corners(t1, 2);
    s1 = h_corners(t1, 3);
    t1 = h_corners(t1, 4);

    x2 = h_corners(t2, 1);
    y2 = h_corners(t2, 2);
    s2 = h_corners(t2, 3);
    t2 = h_corners(t2, 4);

    x3 = h_corners(t3, 1);
    y3 = h_corners(t3, 2);
    s3 = h_corners(t3, 3);
    t3 = h_corners(t3, 4);

    x4 = h_corners(t4, 1);
    y4 = h_corners(t4, 2);
    s4 = h_corners(t4, 3);
    t4 = h_corners(t4, 4);

    temp1  = [-x1 -y1 -1 0 0 0 s1*x1 s1*y1
              0 0 0 -x1 -y1 -1 t1*x1 t1*y1
              -x2 -y2 -1 0 0 0 s2*x2 s2*y2
              0 0 0 -x2 -y2 -1 t2*x2 t2*y2
              -x3 -y3 -1 0 0 0 s3*x3 s3*y3
              0 0 0 -x3 -y3 -1 t3*x3 t3*y3
             -x4 -y4 -1 0 0 0 s4*x4 s4*y4
              0 0 0 -x4 -y4 -1 t4*x4 t4*y4 ];

    temp2 = [-s1 -t1 -s2 -t2 -s3 -t3 -s4 -t4]';


    if (not(inv(temp1)==inf))

        totald = 0;

        hmp = inv(temp1)*temp2;

        tmp_h_matrix = [hmp(1) hmp(2) hmp(3)
                        hmp(4) hmp(5) hmp(6)
                        hmp(7) hmp(8) 1 ];
```

```
        for i = 1:nomatchedcorners

            ix = matchedcorners(i, 1);
            iy = matchedcorners(i, 2);
            fx = matchedcorners(i, 3);
            fy = matchedcorners(i, 4);

            newxyz = tmp_h_matrix*[matchedcorners(i, 1:2) 1]';
            ex = newxyz(1);
            ey = newxyz(2);

            p = ((ix-ex)^2 + (iy-ey)^2)^.5;
            q = ((fx-ix)^2 + (fy-iy)^2)^.5;
            r = ((fx-ex)^2 + (fy-ey)^2)^.5;

            t = (acos((p^2 + q^2 - r^2) / (2*p*q))) * 180/pi;
            d = r*100/q;

            totald = totald + d;

        end

        if totald < mintotald
            h_matrix = tmp_h_matrix;
        end

    end
end

relcorners = [];
temp1 = [];
temp2 = [];

% Finding inliers for best homography matrix

for i = 1:nomatchedcorners

    ix = matchedcorners(i, 1);
    iy = matchedcorners(i, 2);
    fx = matchedcorners(i, 3);
    fy = matchedcorners(i, 4);
    newxyz = h_matrix*[matchedcorners(i, 1:2) 1]';
    ex = newxyz(1);
    ey = newxyz(2);

    p = ((ix-ex)^2 + (iy-ey)^2)^.5;
    q = ((fx-ix)^2 + (fy-iy)^2)^.5;
    r = ((fx-ex)^2 + (fy-ey)^2)^.5;
    t = (acos((p^2 + q^2 - r^2) / (2*p*q))) * 180/pi;
    d = r*100/q;

    if t<25 & d<40 & r<25

        relcorners = [relcorners; ix iy fx fy];

        temp1 = [temp1; -ix -iy -1 0 0 0 fx*ix fx*iy; 0 0 0 -ix -iy -1
fy*ix fy*iy];
        temp2 = [temp2; -fx; -fy]

    end
```

```
end

% Re-estimating H with inliers

hmp = inv(temp1)*temp2;

h_matrix = [hmp(1) hmp(2) hmp(3)
            hmp(4) hmp(5) hmp(6)
            hmp(7) hmp(8) 1 ];



% Computing new warped corners for each correspondence with new H

cornerswrp = [];
for i = 1:nomatchedcorners

    ix = matchedcorners(i, 1);
    iy = matchedcorners(i, 2);
    fx = matchedcorners(i, 3);
    fy = matchedcorners(i, 4);
    newxyz = h_matrix*[corners(i, 1:2) 1]';
    ex = newxyz(1);
    ey = newxyz(2);
    cornerswrp = [cornerswrp; ix iy fx fy ex ey];

end


% Displaying corner correspondences with respective warped corners
% Inliers used to compute H indicated as circles

figure, imagesc(image1), axis image, colormap(gray), hold on
title('Initial Image - Initial, Final and Warped Corners');
for i=1:nomatchedcorners
    plot(cornerswrp(i, [2 4]), cornerswrp(i, [1 3]), 'g-');
    plot(cornerswrp(i, [2 6]), cornerswrp(i, [1 5]), 'g-');
end
plot(cornerswrp(1:r,2), cornerswrp(1:r,1), 'r.');
plot(cornerswrp(1:r,4), cornerswrp(1:r,3), 'b+');
plot(cornerswrp(1:r,6), cornerswrp(1:r,5), 'g^');


figure, imagesc(image1), axis image, colormap(gray), hold on
title('Final Image - Initial, Final and Warped Corners');
for i=1:nomatchedcorners
    plot(cornerswrp(i, [2 4]), cornerswrp(i, [1 3]), 'g-');
    plot(cornerswrp(i, [2 6]), cornerswrp(i, [1 5]), 'g-');
end
plot(cornerswrp(1:r,2), cornerswrp(1:r,1), 'r.');
plot(cornerswrp(1:r,4), cornerswrp(1:r,3), 'b+');
plot(cornerswrp(1:r,6), cornerswrp(1:r,5), 'g^');


if warp == 1

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Obstacle Detection via Warping %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
imagew=zeros(row, col);
plottedw=image3;

% Forming warped image using H and image1

for x=1:row
    for y=1:col

        newxyz= h_matrix*[x y 1];
        s = newxyz(1);
        t = newxyz(1);


        if s>=1 & t>=1 & s<=row & t<=col
            image3(s,t) =
(image3(s,t)*plottedw(s,t)+image1(x,y))/(plottedw(s,t)+1);
            plottedw(s,t) = plottedw(s,t)+1;
        end

    end
end


% Filling out blanks in warped image averaging plotted neighbours

notplotted = 1;

while notplotted>0

    notplotted = 0;
    for x = 2:row-1
        for y = 2:col-1

        nbrs_pltd = 0;
            nbrs_sumd = 0;
            if plottedw(x,y) == 0

                  if plottedw(x-1,y-1)>0
                        nbrs_sumd = nbrs_sumd + image3(x-1,y-1);
                        nbrs_pltd = nbrs_pltd + 1;
                  end
                  if plottedw(x,y-1)>0
                        nbrs_sumd = nbrs_sumd + image3(x,y-1);
                        nbrs_pltd = nbrs_pltd + 1;
                  end
                  if plottedw(x+1,y-1)>0
                        nbrs_sumd = nbrs_sumd + image3(x+1,y-1);
                        nbrs_pltd = nbrs_pltd + 1;
                  end
                  if plottedw(x+1,y)>0
                        nbrs_sumd = nbrs_sumd + image3(x+1,y);
                        nbrs_pltd = nbrs_pltd + 1;
                  end
                  if plottedw(x+1,y+1)>0
                        nbrs_sumd = nbrs_sumd + image3(x+1,y+1);
                        nbrs_pltd = nbrs_pltd + 1;
                  end
                  if plottedw(x,y+1)>0
                        nbrs_sumd = nbrs_sumd + image3(x,y+1);
                        nbrs_pltd = nbrs_pltd + 1;
                  end
```

```
                if plottedw(x-1,y+1)>0
                      nbrs_sumd = nbrs_sumd + image3(x-1,y+1);
                      nbrs_pltd = nbrs_pltd + 1;
                end
                if plottedw(x-1,y+1)>0
                      nbrs_sumd = nbrs_sumd + image3(x-1,y+1);
                      nbrs_pltd = nbrs_pltd + 1;
                end

                if nbrs_sumd > 0
                      image3(x,y) = nbrs_sumd/nbrs_pltd;
              plottedw(x,y) = 1;
                else
                      notplotted = notplotted+1;
                end

            end
        end
    end
end


imagec=zeros(row, col);
blockdiff=zeros(30,40);

% Blocks indicated as obstacles if corner in block is above threshold with
% comparison to its warped corner

for i = 1:nomatchedcorners

    ix = matchedcorners(i, 1);
    iy = matchedcorners(i, 2);
    fx = matchedcorners(i, 3);
    fy = matchedcorners(i, 4);

    newxyz = h_matrix*[matchedcorners(i, 1:2) 1]';
    ex = newxyz(1);
    ey = newxyz(2);


    p = ((ix-ex)^2 + (iy-ey)^2)^.5;
    q = ((fx-ix)^2 + (fy-iy)^2)^.5;
    r = ((fx-ex)^2 + (fy-ey)^2)^.5;
    t = (acos((p^2 + q^2 - r^2) / (2*p*q))) * 180/pi;
    d = r*100/q;

    if t<25 & d<40 & r<25
       blockdiff((ex-mod(ex,16))/16+1, (ey-mod(ey,16))/16+1) = 10000;
    end

end


% Blocks indicated as obstacles if sum difference of pixels between block
% on final image and warped image are above threshold

for i=1:30
  for j=1:40

      blockdiff(i,j) = sum(sum(double(image2((i-1)*16+1:i*16, (j-
1)*16+1:j*16)) - double(image3((i-1)*16+1:i*16, (j-1)*16+1:j*16))));
```

```
      if blockdiff(i,j) < 10000
          imagec((i-1)*16+1:i*16-1,(j-1)*16+1:j*16-1) = image2((i-
1)*16+1:i*16-1,(j-1)*16+1:j*16-1);
      end
  end
end

% Difference Image between final image and warped image
imagex = abs(double(imagew)-double(image2));
figure, imagesc(imagex), axis image, colormap(gray), hold on
title('Difference between Final Image and Warped Image');

% Blocks showing obstacles via comparison
figure, imagesc(imagec), axis image, colormap(gray), hold on
title('Obstacle Detection - Comparing Warped Image and Final image');

end
```

```
if segmenter == 1

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Obstacle Detection via Segmentation %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

segments = zeros(row,col);
segments(1,1) = 1;
nextsegment = 2;

% Pixels assigned same segment as neighbour with closest intensity

for i=2:col
    if abs(im(1,i) - im(1,i-1)) < 15
        segments(1,i) = segments(1,i-1);
    else
        segments(1,i) = nextsegment;
        nextsegment = nextsegment + 1;
    end
end

for i=2:row
    if abs(im(i,1) - im(i-1,1)) < 15
        segments(i,1) = segments(i-1,1);
    else
        segments(i,1) = nextsegment;
        nextsegment = nextsegment + 1;
    end
end

for i=2:row
    for j=2:col

        as = abs(im(i,j) - (im(i,j-1)));
        bs = abs(im(i,j) - (im(i-1, j)));
        cs = abs(im(i,j) - (im(i-1,j-1)));
```

```
        if min([as bs cs]) <= 25
            if as <= bs & as <= cs
                segments(i,j) = segments(i,j-1);
            else
                if bs <= as & bs <= cs
                    segments(i,j) = segments(i-1,j);
                else
                    if cs <= as & cs <= bs
                        segments(i,j) = segments(i-1,j-1);
                    else
                        segments(i,j) = 200;
                    end
                end
            end
        else
            segments(i,j) = nextsegment;
            nextsegment = nextsegment + 1;
        end

    end
end

segchanges = 1;
segs = nextsegment-1

% Adjacent segments with similar intensities merged together

while segchanges>0
    for x=1:row-1
        for y=1:col-1

            if im(x,y)-im(x+1,y)<15 and not(segments(x,y)==segments(x+1,y)
                segchanges = 1;
                segtochange = segments(x+1,y);
                for i=1:row
                    for j=1:col
                        if segments(i,j)==segtochange
                            segments(i,j)=segments(x,y);
                        end
                    end
                end
            end

            if im(x,y)-im(x,y+1)<15 and not(segments(x,y)==segments(x,y+1)
                segchanges = 1;
                segtochange = segments(x,y+1);
                for i=1:row
                    for j=1:col
                        if segments(i,j)==segtochange
                            segments(i,j)=segments(x,y);
                        end
                    end
                end
            end

        end
    end
end

segheights = zeros(segs, 1);
```

```
% Height of each corner computed

cheights = [];
for i = 1:nomatchedcorners

    ix = matchedcorners(i, 1);
    iy = matchedcorners(i, 2);
    fx = matchedcorners(i, 3);
    fy = matchedcorners(i, 4);
    ht = calculate_height(ix, iy, fx, fy, h_matrices);
    cheights = [cheights; ht];

end

figure, imagesc(image2), axis image, colormap(gray), hold on
title('Final Image - Heights of Corners');
for i=1:matchedcorners
    text(matchedcorners(i,4), matchedcorners(i,3), num2str(cheights(i)));
end



% Height of each segment computed as maximum height among corners in it

for i=1:nomatchedcorners

    x = corners(i, 3);
    y = corners(i, 4);

    if heights(i) > segheights(segments(x,y))
        segheights(segments(x,y)) = heights(i)
    end

end

pointheight = zeros(row,col);

for x = 1:row
    for y= 1:col

        pointheight(x,y) = segheights(segments(x,y));

    end
end

% Height image formed representing increasing altitude with increasing
% intensity (White = Camera Height)

heightimage = heightimage*255/ch

figure, imagesc(heightimage), axis image, colormap(gray), hold on
title('Obstacle Detection - Segmentation');

end


if edger==1

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Obstacle Detection via Contour Heights %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
matchL = [];
matchR = [];

for i=1:matchedcorners

    matchL = [matchL; corners(i,1:2) 1];
    matchR = [matchR; corners(i,3:4) 1];

end

matchL = matchL';
matchR = matchR';

% F Matrix Computation using RANSAC
[f_matrix goodL goodR x] = fundamental_RANSAC(matchL,matchR);


% Matching edge points found using epipolar lines

pointtotals = [];

for x=1:row
    for y=1:col

        if edgeim1(x,y)==1
          xedges = xedges+1;
          if mod(xedges,4) == 0

            lm = -(fa*x + fd*y + fg) / (fb*x + fe*y + fh);
            lc = -(fc*x + ff*y + fi) / (fb*x + fe*y + fh);

            % equation of epipolar line j = lm*i + lc

            if lm < 20 then

                  for i = x-20:x+20

                        j = lm * i + lc;

                    mintotal = 30000;

                    if edgeim2(i,j)== 1

                        difftotal = sum(
abs{image1(x-nhood:x+nhood, y-nhood:y+nhood)  - image2(i-nhood:i+nhood, j-
nhood:j+nhood)} )

                          if difftotal<mintotal
                                    mintotal = cortotal;
                                    tempi = i;
                                    tempj = j;
                                    found = 1;
                               end
                        end

                  end

            else

                for j = y-20:y+20
```

```
                        i = (j - lc) / lm;

                    mintotal = 30000;

                    if edgeim2(i,j) == 1
                                difftotal = sum( abs{image1(x-
nhood:x+nhood, y-nhood:y+nhood) - image2(i-nhood:i+nhood, j-
nhood:j+nhood)})

                            if difftotal<mintotal
                                  mintotal = diffcortotal;
                                  tempi = i;
                                  tempj = j;
                                  found = 1;
                            end
                        end

                end
            end

            if found==1

                pointtotals = [pointtotals; x y tempi tempj mintotal];

            end
          end
        end
      end

end

[nomatchedpoints tmp] = size(pointtotals);
matchedpoints = [];

% Mathcedpoints turned into one-one relation such one point on image1
% corresponds to just one point in image2 and vice versa

for u=1:nomatchedtotals

    i = pointtotals(i,3);
    j = pointtotals(i,4);

    mintotal = 65535;

    for v=1:nomatchedpoints


        a = cornertotals(v,1);
        b = cornertotals(v,2);
        p = cornertotals(v,3);
        q = cornertotals(v,4);
        dtotal = cornertotals(v,5);


        if  i==p & j==q & dtotal < mintotal

            tempx = a;
            tempy = b;
            mintotal = dtotal;
```

```
        end

    end

    matchedpoints = [matchedpoints; tempx tempy i j];

end

[nomatchedpoints tmp] = size(matchedpoints);
blockheights = zeros(40,40);

figure, imagesc(image2), axis image, colormap(gray), hold on
title('Final Image - Heights along contours');
[r c] = size(fedges);
eheights = [];

% Heights of each edge point computed
% Heights displayed as text on image
% Heights of each block computed as maximum height among points in each
% block

for i = 1:nomatchedpoints

   ix = matchedcorners(i, 1);
   iy = matchedcorners(i, 2);
   fx = matchedcorners(i, 3);
   fy = matchedcorners(i, 4);

   ht = calculate_height(ix, iy, fx, fy, h_matrices);

   if mod(i,20)==0 & mod(fedges(i,2),2)== 0
        text(fedges(i,2)+2, fedges(i,1)-2, num2str(eheights(i)));
   end

   dr = (fx-mod(fx,16))/16+1;
   dc = (fy-mod(fy,16))/16+1;
   if blockheights(dr,dc) < ht
        blockheights(dr,dc)=ht;
   end

end

imageb=imageb+255*imagee3;

for i=1:30
    for j=1:40
        imagebl((i-1)*16+1:i*16-1,(j-1)*16+1:j*16-
1)=255/ch*blockheights(i,j);
    end
end

% Heights of blocks shown
figure, imagesc(imagebl), axis image, colormap(gray), hold on
title('Heights along contours');

end


function ht=calculate_height(ix,iy,fx,fy,h_matrices)

    % Computes height of point correspondence (ix,iy),(fx,fy)
```

```
    % Given a set of homographies for each plane

    htval = 1000;
    ht = 0;

    for h = 0:ch

        hh_matrix = h_matrices(h*3+1:h*3+3, :);

        ix = double(matchedcorners(i,1));
        iy = double(matchedcorners(i,2));
        fx = double(matchedcorners(i,3));
        fy = double(matchedcorners(i,4));
        newxyz = hh_matrix*[ix iy 1]';
        newxy = uint16(newxyz(1:2));
        ex = double(newxy(1));
        ey = double(newxy(2));

        p = ((ix-ex)^2 + (iy-ey)^2)^.5;
        q = ((fx-ix)^2 + (fy-iy)^2)^.5;
        r = ((fx-ex)^2 + (fy-ey)^2)^.5;

        t = acos((p^2 + q^2 - r^2) / (2*p*q));
        t = t * 180 / pi;

        % Computing correspondence between plane and point

        d = double(r)*100/q;

        % Selecting plane with the best correspondence
        if d < htval
            htval = d;
            ht = h;
        end
    end

return



%-------------------------------------
% kmean_seg - Segment an image by K-mean clustering method.
%
% Usage:  edge_seg=kmean_seg(im)
%
% Argument:
%   im - input image
% Returns:
%   colour_seg - image with colour segmentation using k-means
%   edge_seg - the edge of the index image, which is the result of
%   partitions the points in the image into k clusters.
%   The binary image has the same size as original input image,
%   with 1's where the function finds edges and 0's elsewhere.
%
% Zezhi Chen
% Department of Computer Science
% The University of York
% http://www.cs.york.ac.uk/~chen
% January 2005
%
%-------------------------------------
```

```
function [colour_seg edge_seg]=kmean_seg(im)
CI1=wiener2(im(:,:,1),[7,7]);
CI2=wiener2(im(:,:,2),[7,7]);
CI3=wiener2(im(:,:,3),[7,7]);
newhe(:,:,1)=CI1;
newhe(:,:,2)=CI2;
newhe(:,:,3)=CI3;

% RGB clour space, the result of this space is better
ab = double(newhe);
nrows = size(ab,1);
ncols = size(ab,2);
ab = reshape(ab,nrows*ncols,3);

nColors = 3;
% repeat the clustering 3 times to avoid local minima
[cluster_idx cluster_center] = kmeans(ab,nColors,'distance','sqEuclidean',
...
    'Replicates',3);
pixel_labels = reshape(cluster_idx,nrows,ncols);
labelcolor=label2rgb(pixel_labels);

area=80;
for ci=1:nColors
    newpixel_labels=pixel_labels;
    class1=newpixel_labels;
    ind1=find(class1~=ci);
    class1(ind1)=0;
    L1 = bwlabeln(class1);
    S1 = regionprops(L1, 'Area', 'PixelList');
    Areaindx=find([S1.Area] < area);
    for IR=1:length(Areaindx);
        holeidx=S1(Areaindx(IR)).PixelList;
        seeds=max(holeidx,[],1)+1;
        if (seeds(1)<ncols)&(seeds(2)<nrows)
            seedcolor=pixel_labels(seeds(2),seeds(1));
            for kk=1:size(holeidx,1)
                newpixel_labels(holeidx(kk,2),holeidx(kk,1))=seedcolor;
            end
        end
    end
end
newlabelcolor=label2rgb(newpixel_labels);

gray_pixel_labels=newpixel_labels/max(max(newpixel_labels));
colour_seg=medfilt2(gray_pixel_labels,[7,7]);
edge_seg=edge(colour_seg,'canny');
return



%---------------------------
%this program is to compute Fundamental Matrix and matching points by using
improved 8-point algorithm and RANSAC method
%
% Usage:   [matrix, goodmatchL, goodmatchR, residual] =
fundamental_RANSAC(matchL,matchR)
%
% Arguments:
% matchL - 3xm homogenous imagecoordinates (x,y,1)'
```

```
% matchR - 3xn homogenous imagecoordinates (x',y',1)', such that  matchL<-
>matchR
% maxloop - The number of maximum iter for RANSAC method, usually let
% maxloop==5000
%
% Returns:
%          matrix - Fundamental matrix such that matchR'*matrix*matchL=0
%          goodmatchL - Good matching points of the left image after RANSAC
%          goodmatchR - Good matching points of the right image after
RANSAC
%          resid - Average residual of good matching pairs


%----------------------------
%date: April 23,2004
% The University of York
%Chen Zezhi
%----------------------------

function [matrix, goodmatchL, goodmatchR, resid] =
fundamental_RANSAC(matchL,matchR,maxloop)

matchnumber=size(matchL,2);
xm=mean(matchL');
ym=mean(matchR');

% Attempt to normalise each set of points so that the origin
% is at centroid and mean distance from origin is sqrt(2).
  [corL, t1] = normalise2dpts(matchL);
  [corR, t2] = normalise2dpts(matchR);

colL=size(corL,2);
colR=size(corR,2);
eachL=fix(colL/8);
eachR=fix(colR/8);
matchL1=[];
matchL2=[];
matchL3=[];
matchL4=[];
matchL5=[];
matchL6=[];
matchL7=[];
matchL8=[];
matchL1=[matchL1 [corL(:,1:eachL)]];
matchL2=[matchL2 [corL(:,eachL+1:2*eachL)]];
matchL3=[matchL3 [corL(:,2*eachL+1:3*eachL)]];
matchL4=[matchL4 [corL(:,3*eachL+1:4*eachL)]];
matchL5=[matchL5 [corL(:,4*eachL+1:5*eachL)]];
matchL6=[matchL6 [corL(:,5*eachL+1:6*eachL)]];
matchL7=[matchL7 [corL(:,6*eachL+1:7*eachL)]];
matchL8=[matchL8 [corL(:,7*eachL+1:8*eachL)]];
if colL-eachL*8-1>=0
matchL1=[matchL1 [corL(:,8*eachL+1)]];
end
if colL-eachL*8-2>=0
matchL2=[matchL2 [corL(:,8*eachL+2)]];
end
if colL-eachL*8-3>=0
matchL3=[matchL3 [corL(:,8*eachL+3)]];
end
if colL-eachL*8-4>=0
```

```
matchL4=[matchL4 [corL(:,8*eachL+4)]];
end
if colL-eachL*8-5>=0
matchL5=[matchL5 [corL(:,8*eachL+5)]];
end
if colL-eachL*8-6>=0
matchL6=[matchL6 [corL(:,8*eachL+6)]];
end
if colL-eachL*8-7>=0
matchL7=[matchL7 [corL(:,8*eachL+7)]];
end

matchR1=[];
matchR2=[];
matchR3=[];
matchR4=[];
matchR5=[];
matchR6=[];
matchR7=[];
matchR8=[];
matchR1=[matchR1 [corR(:,1:eachL)]];
matchR2=[matchR2 [corR(:,eachL+1:2*eachL)]];
matchR3=[matchR3 [corR(:,2*eachL+1:3*eachL)]];
matchR4=[matchR4 [corR(:,3*eachL+1:4*eachL)]];
matchR5=[matchR5 [corR(:,4*eachL+1:5*eachL)]];
matchR6=[matchR6 [corR(:,5*eachL+1:6*eachL)]];
matchR7=[matchR7 [corR(:,6*eachL+1:7*eachL)]];
matchR8=[matchR8 [corR(:,7*eachL+1:8*eachL)]];
if colR-eachL*8-1>=0
matchR1=[matchR1 [corR(:,8*eachL+1)]];
end
if colR-eachR*8-2>=0
matchR2=[matchR2 [corR(:,8*eachL+2)]];
end
if colR-eachR*8-3>=0
matchR3=[matchR3 [corR(:,8*eachL+3)]];
end
if colR-eachR*8-4>=0
matchR4=[matchR4 [corR(:,8*eachL+4)]];
end
if colR-eachR*8-5>=0
matchR5=[matchR5 [corR(:,8*eachL+5)]];
end
if colR-eachR*8-6>=0
matchR6=[matchR6 [corR(:,8*eachL+6)]];
end
if colR-eachR*8-7>=0
matchR7=[matchR7 [corR(:,8*eachL+7)]];
end

% get better initial value
threshold=10;
mindis=inf;
K=0;
while (1)
listL=fix(rand(1,8)*eachL+1);
listR=fix(rand(1,8)*eachR+1);
FF=[matchR1(1,listR(1))*matchL1(1,listL(1))
matchR1(2,listR(1))*matchL1(1,listL(1)) matchL1(1,listL(1))
matchR1(1,listR(1))*matchL1(2,listL(1))
```

```
matchR1(2,listR(1))*matchL1(2,listL(1)) matchL1(2,listL(1))
matchR1(1,listR(1)) matchR1(2,listR(1)) 1
    matchR2(1,listR(2))*matchL2(1,listL(2))
matchR2(2,listR(2))*matchL2(1,listL(2)) matchL2(1,listL(2))
matchR2(1,listR(2))*matchL2(2,listL(2))
matchR2(2,listR(2))*matchL2(2,listL(2)) matchL2(2,listL(2))
matchR2(1,listR(2)) matchR2(2,listR(2)) 1
    matchR3(1,listR(3))*matchL3(1,listL(3))
matchR3(2,listR(3))*matchL3(1,listL(3)) matchL3(1,listL(3))
matchR3(1,listR(3))*matchL3(2,listL(3))
matchR3(2,listR(3))*matchL3(2,listL(3)) matchL3(2,listL(3))
matchR3(1,listR(3)) matchR3(2,listR(3)) 1
    matchR4(1,listR(4))*matchL4(1,listL(4))
matchR4(2,listR(4))*matchL4(1,listL(4)) matchL4(1,listL(4))
matchR4(1,listR(4))*matchL4(2,listL(4))
matchR4(2,listR(4))*matchL4(2,listL(4)) matchL4(2,listL(4))
matchR4(1,listR(4)) matchR4(2,listR(4)) 1
    matchR5(1,listR(5))*matchL5(1,listL(5))
matchR5(2,listR(5))*matchL5(1,listL(5)) matchL5(1,listL(5))
matchR5(1,listR(5))*matchL5(2,listL(5))
matchR5(2,listR(5))*matchL5(2,listL(5)) matchL5(2,listL(5))
matchR5(1,listR(5)) matchR5(2,listR(5)) 1
    matchR6(1,listR(6))*matchL6(1,listL(6))
matchR6(2,listR(6))*matchL6(1,listL(6)) matchL6(1,listL(6))
matchR6(1,listR(6))*matchL6(2,listL(6))
matchR6(2,listR(6))*matchL6(2,listL(6)) matchL6(2,listL(6))
matchR6(1,listR(6)) matchR6(2,listR(6)) 1
    matchR7(1,listR(7))*matchL7(1,listL(7))
matchR7(2,listR(7))*matchL7(1,listL(7)) matchL7(1,listL(7))
matchR7(1,listR(7))*matchL7(2,listL(7))
matchR7(2,listR(7))*matchL7(2,listL(7)) matchL7(2,listL(7))
matchR7(1,listR(7)) matchR7(2,listR(7)) 1
    matchR8(1,listR(8))*matchL8(1,listL(8))
matchR8(2,listR(8))*matchL8(1,listL(8)) matchL8(1,listL(8))
matchR8(1,listR(8))*matchL8(2,listL(8))
matchR8(2,listR(8))*matchL8(2,listL(8)) matchL8(2,listL(8))
matchR8(1,listR(8)) matchR8(2,listR(8)) 1];

[s,v,d]=svd(FF'*FF);
xs=d(:,9);
f=[xs(1) xs(2) xs(3);xs(4) xs(5) xs(6);xs(7) xs(8) xs(9)];
f=f';

% Enforce constraint that fundamental matrix has rank 2 by performing
% a svd and then reconstructing with the two largest singular values.
    [U,D,V] = svd(f);
    F = U*diag([D(1,1) D(2,2) 0])*V';
f1=t2'*F*t1;
f2=f1';
K=K+1;
dist1=0;
for i=1:matchnumber
xl=[matchL(1,i) matchL(2,i) 1]';
f1xl=f1*xl;
xr=[matchR(1,i) matchR(2,i) 1]';
f2xr=f2*xr;
dis1=abs(xr'*f1*xl)/sqrt(f1xl(1)*f1xl(1)+f1xl(2)*f1xl(2));
dis2=abs(xl'*f2*xr)/sqrt(f2xr(1)*f2xr(1)+f2xr(2)*f2xr(2));
dist1=dist1+dis1+dis2;
end
if dist1<mindis
```

```
    mindis=dist1;
end
    if K>5000
       disp(sprintf('The minimal average epipolar distance is
%d',mindis/(2*matchnumber)));
        disp('It is not good data');
        matrix=[];
        resid=inf;
        goodmatchL=[];
        goodmatchR=[];
       return
    elseif (mindis/(2*matchnumber))<threshold
    disp('It is good data');

% get good matching points by using initial value
goodmatchL=[];
goodmatchR=[];
badmatchL=[];
badmatchR=[];
distance=0;
for i=1:matchnumber
xl=[matchL(1,i) matchL(2,i) 1]';
f1xl=f1*xl;
xr=[matchR(1,i) matchR(2,i) 1]';
f2xr=f2*xr;
dis1=abs(xr'*f1*xl)/sqrt(f1xl(1)*f1xl(1)+f1xl(2)*f1xl(2));
dis2=abs(xl'*f2*xr)/sqrt(f2xr(1)*f2xr(1)+f2xr(2)*f2xr(2));
distance=dis1+dis2;
if distance<(2*threshold)
    goodmatchL=[goodmatchL [xl]];
    goodmatchR=[goodmatchR [xr]];
else
    badmatchL=[badmatchL [xl]];
    badmatchR=[badmatchR [xr]];
end
end
  break;
end
end

% compute fundamental matrix by using good matches
m=goodmatchL;
mp=goodmatchR;
xm=mean(m');
ym=mean(mp');

% Attempt to normalise each set of points so that the origin
% is at centroid and mean distance from origin is sqrt(2).
 [m, t1] = normalise2dpts(goodmatchL);
 [mp, t2] = normalise2dpts(goodmatchR);


Number=size(goodmatchL,2);
    disp(sprintf('There are %d good matching points', Number));

if Number>=8
for j=8:Number

       a=[];

       for i=1:j,
```

```
        a=[a;[mp(1,i)*m(1,i) mp(2,i)*m(1,i) m(1,i) mp(1,i)*m(2,i)
mp(2,i)*m(2,i) m(2,i) mp(1,i) mp(2,i) 1]];
       end
[s,v,d]=svd(a'*a);
xs=d(:,9);
fs=[xs(1) xs(2) xs(3);xs(4) xs(5) xs(6);xs(7) xs(8) xs(9)];
fs=fs';
% Enforce constraint that fundamental matrix has rank 2 by performing
% a svd and then reconstructing with the two largest singular values.
    [U,D,V] = svd(fs);
    Fs = U*diag([D(1,1) D(2,2) 0])*V';
f1=t2'*Fs*t1;
f2=f1';

rest=0;
distance=0;
mindist=inf;
for i=1:Number
xl=[goodmatchL(1,i) goodmatchL(2,i) 1]';
f1xl=f1*xl;
xr=[goodmatchR(1,i) goodmatchR(2,i) 1]';
f2xr=f2*xr;
dis1=abs(xr'*f1*xl)/sqrt(f1xl(1)*f1xl(1)+f1xl(2)*f1xl(2));
dis2=abs(xl'*f2*xr)/sqrt(f2xr(1)*f2xr(1)+f2xr(2)*f2xr(2));
distance=distance+dis1+dis2;

end
if distance<mindist
   mindist=distance;
  fundf=f1;
end
end
ff=fundf(:);
ss=norm(ff,'fro');
matrix=fundf/ss;
 f1=matrix;
f2=f1';
distance=0;
for i=1:Number
xl=[goodmatchL(1,i) goodmatchL(2,i) 1]';
f1xl=f1*xl;
xr=[goodmatchR(1,i) goodmatchR(2,i) 1]';
f2xr=f2*xr;
dis1=abs(xr'*f1*xl)/sqrt(f1xl(1)*f1xl(1)+f1xl(2)*f1xl(2));
dis2=abs(xl'*f2*xr)/sqrt(f2xr(1)*f2xr(1)+f2xr(2)*f2xr(2));
distance=distance+dis1+dis2;
end
resid=distance/(2*Number);

else
    disp('Sorry, not enough good matching points');
end
return




% HARRIS - Harris corner detector
%
% Usage:  [cim, r, c] = harris(im, sigma, thresh, radius, disp)
```

```
%
% Arguments:
%            im     - image to be processed.
%            sigma  - standard deviation of smoothing Gaussian. Typical
%                     values to use might be 1-3.
%            thresh - threshold (optional). Try a value ~1000.
%            radius - radius of region considered in non-maximal
%                     suppression (optional). Typical values to use might
%                     be 1-3.
%            disp   - optional flag (0 or 1) indicating whether you want
%                     to display corners overlayed on the original
%                     image. This can be useful for parameter tuning.
%
% Returns:
%            cim    - binary image marking corners.
%            r      - row coordinates of corner points.
%            c      - column coordinates of corner points.
%
% If thresh and radius are omitted from the argument list 'cim' is returned
% as a raw corner strength image and r and c are returned empty.

% Reference:
% C.G. Harris and M.J. Stephens. "A combined corner and edge detector",
% Proceedings Fourth Alvey Vision Conference, Manchester.
% pp 147-151, 1988.
%
% Author:
% Peter Kovesi
% Department of Computer Science & Software Engineering
% The University of Western Australia
% pk@cs.uwa.edu.au  www.cs.uwa.edu.au/~pk
%
% March 2002

function [cim, r, c] = harris(im, sigma, thresh, radius, disp)

    error(nargchk(2,5,nargin));

    dx = [-1 0 1; -1 0 1; -1 0 1]; % Derivative masks
    dy = dx';

    Ix = conv2(im, dx, 'same');    % Image derivatives
    Iy = conv2(im, dy, 'same');

    % Generate Gaussian filter of size 6*sigma (+/- 3sigma) and of
    % minimum size 1x1.
    g = fspecial('gaussian',max(1,fix(6*sigma)), sigma);

    Ix2 = conv2(Ix.^2, g, 'same'); % Smoothed squared image derivatives
    Iy2 = conv2(Iy.^2, g, 'same');
    Ixy = conv2(Ix.*Iy, g, 'same');

    cim = (Ix2.*Iy2 - Ixy.^2)./(Ix2 + Iy2 + eps); % Harris corner measure

    % Alternate Harris corner measure used by some.  Suggested that
    % k=0.04 - I find this a bit arbitrary and unsatisfactory.
%   cim = (Ix2.*Iy2 - Ixy.^2) - k*(Ix2 + Iy2).^2;

    if nargin > 2   % We should perform nonmaximal suppression and
threshold
```

```
    % Extract local maxima by performing a grey scale morphological
    % dilation and then finding points in the corner strength image that
    % match the dilated image and are also greater than the threshold.
    sze = 2*radius+1;                   % Size of mask.
    mx = ordfilt2(cim,sze^2,ones(sze)); % Grey-scale dilate.
    cim = (cim==mx)&(cim>thresh);       % Find maxima.

    [r,c] = find(cim);                  % Find row,col coords.

    if nargin==5 & disp      % overlay corners on original image
%         figure, imagesc(im), axis image, colormap(gray), hold on
%        axis off
%         plot(c,r,'r+'), title('corners detected');
    end

  else  % leave cim as a corner strength image and make r and c empty.
    r = []; c = [];
  end
```