LEARNING VISUAL ODOMETRY

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

October 2006

By
Franck Bettinger
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Biological systems use visual information in many different contexts. One example is the ability to focus attention, another the ability to use visual information as a navigational aid. As navigation is an important issue in the field of robotics, it is worth trying to build robots that are able to mimic the natural computation of visual information. This project examines and implements mechanisms to develop these behaviours on a mobile robot. A corridor following algorithm was implemented which used visual information to follow the corridor. It worked successfully in a number of different corridors.

Another point that is important in navigation is the ability to perceive distances. Indeed, distances are usually taken as parameters when a plan of the environment is created, either by an operator or by the robot itself. In this project a learning mechanism is used to learn the relationship between optic flow and ego-motion: the robot learns to estimate its own motion, judging from changes in the image. The robot is able to estimate distances with less than 10% relative error, as long as the travelled distance is sufficiently long. For small distances the relative error increases.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

I would like to thank my supervisor, Dr Ulrich Nehmzow and his PhD student, Stephen Marsland for their guidance and their useful advice.

I would also like to thank all the members of the Robotics group for their kindness and their help, especially Andrew Pickering for his assistance regarding the hardware.

And finally, I would like to thank Dr David Rydeheard for his invaluable advice on the successful completion of a MSc project.

Dedicace

I dedicate this thesis to my family.

Chapter 1

Introduction

1.1 Motivation for this work

Since its beginning, the robotics field had tried to increase the interaction between robots and their environment. The first robots were only able to perform repetitive tasks. These robots were hardwired so there was no possibility of changing their behaviour. In order to introduce interaction between humans (such as the programmer) and the robot, people introduced robots that were able to be reprogrammed.

The problem with these robots is that they must be programmed for each new situation. Indeed, a programmer cannot design all the possible situations. This is especially true for robots which have to operate in unconstrained environments where people move about too.

To be efficient, a robot must interact with its environment. It has to be able to react correctly to unknown events that can occur. Learning by experience is then a useful feature we can implant on a robot. In this thesis we are particularly interested in corridor following and egomotion estimation using visual flow. These are two features that are needed to navigate correctly using a map provided by the user.

Egomotion estimation is also needed as feedback for odometry. Indeed, wheel encoders are often not accurate. Imagine the difference between a robot driving a long distance on a road and the same robot driving on the sand. In the first case, the robot is moving forward and the odometry system is approximately recording the right distance. In the second case, the robot just slips, but the odometry system records a long distance. This recorded distance is wrong because it should be approximately zero. The slippage confuses the odometry system because it involves an unknown friction coefficient. This situation is quite frequent in indoors environments too. If a robot is stuck against a wall, its wheels turn but the robot does not move.

On some robots, we cannot use wheel encoders to give us an idea of the distance travelled so far. Basic examples are helicopters, submarines, legged robots and any robots which do not have wheels. In these situations, egomotion estimation can be essential.

1.2 Related work

Usually, when a robot is being built, people tend to use optic flow to control the behaviour of the robot. It is a powerful way to perceive distances from a sequence of images. It can be computed using different algorithms [23, 5, 2]. With optical flow it is possible to build robots able to navigate in complex environments. Obstacle avoidance and corridor following can be done using optic flow. For instance, in [7] the game of tag was programmed on two robots using obstacle avoidance based on optical flow.

Optical flow is also used for automatic road following in vision-based intelligent vehicles [3]. However, optical flow needs a lot of computation. So it requires a lot of processing power which is usually not available on mobile agents. That is why this thesis tries to investigate other methods of navigation which do not require a computation of the optic flow. We want neural networks to estimate the optic flow for us.

Learning methods have already been used a lot in order to mimic animal behaviour. Some of them use optic flow as a preprocessing stage [25, 26], other use simpler versions of optic flow (one dimension optic flow for instance [32]), and other do not use optic flow at all. We will focus our work on the third category, this being the vision of the flies has been modeled with neural networks without using optic flow [13]. Our goal is to build a corridor following robot that is able to learn its task.

Another study about animals concern the visual path integration. Srinivasan has focused his study specifically on bees [28, 29]. This study has lead to construct of a robot that uses of the visual system of a bee. Unfortunately, the implementation uses two cameras [12]. It has been shown that bees can still estimate how far they travelled even if they only use one eye. So our second goal is to estimate the egomotion of a robot by using a learning method and only one camera.

Our aim is to create a new odometry system and use it to validate the existing one. Such validation techniques are widely used in robotics. Sonars are usually used to do this, but any sensors can be used to reduce odometry errors [15, 30]. For instance, the article [24] describes a general method to reduce odometry errors using another sensor.

The following sections will briefly describe the experiments done and the results extracted from these experiments.

1.3 Experimental approach

In order to reach our goals, we preferred an experimental approach. Simulation is not appropriate especially because of the visual input. Visual inputs are hard to simulate properly. Even if we can simulate proper visual inputs by using physics of light for instance, it will require a lot of computation time.

Furthermore, the real world is such complex that is almost impossible to model it correctly. The simulation will suffer from this lack of complexity and, when tested, the robot will perform badly. Moreover, it is also hard to simulate noise. Unfortunately, noise is often a key issue in robotics. Thresholds are often adjusted according to the amount of noise. Therefore it is better to make real experiments instead of simulations.

For this project, experiments were made without using simulation.

1.4 Experiments

Different experiments were done to build a robot that can follow a corridor. For the first version of a corridor following robot, a camera was always pointing towards the direction of the navigation, that is, towards the end of the corridor. This experiment was done with a robot called *FortyTwo* which has the particularity of having a turret that can turn independently of the rest of the robot.

After a change in the hardware of *FortyTwo*, we could not do the any other experiments with it. So another robot called *Charm* was used. The camera was not able to stay in a fixed direction with *Charm*. So we had to change completely our way of performing the corridor following.

The new algorithm done, we had problems to teach the robot how to follow corridor because of the computation time required. So we simply build another corridor following program which was not able to learn but only to follow corridors. We then used this hardwired program to teach the first program how to learn to follow a corridor.

At least, we build a robot that was able to estimate the distance that it travelled. During this experiment, the robot was also self taught. It simply went forward, took pictures of the room and recorded odometry measures during the same time.

1.5 Results

The first corridor follower built using the robot *FortyTwo* was not perfect. This was due to the way the robot was trained. But the results let us hope that it is possible to build a corridor follower using *FortyTwo*. We simply could not test this corridor follower because of the change in the hardware.

The version of the corridor following robot using automatic learning gave good results. The robot was trained in one corridor and tested in three different corridors and it worked successfully. The program designed to teach used ceiling as landmarks, but the program that learns used lights as well as ceiling to do the same task. This brought robustness to the program. During the tests, the travelled paths were very close to their mean path. The original orientation of the robot for successed taken paths was also studied. It reveals that the algorithm is relatively robust to the choice of this angle provided that it is in a fixed range (from $-10°$ to $30°$).

Finally, the distance estimation experiments showed that for long distances it performs better than the internal odometry based on wheels. For shorter distances the internal odometry performs better.

Chapter 2

Mechanisms

2.1 Tools

Two robots were used for this project: *FortyTwo*, a Nomad 200 and *Charm*, a Nomad Scout. All we have to know for now is that both run the Linux operating system and use a frame grabber which is able to take images at a resolution of 640x480 pixels. These robots are described in more detail in sections 3.2.1 and 3.3.1, respectively.

The only inputs used in the experiments performed are a sequences of images grabbed by the CCD camera and the internal odometry of the robot. Internal odometry is only used during a learning stage. So the robot performs its task by using visual sensors and only visual sensors. Furthermore, sonars and infrareds have been used in a safety procedure in order not to damage the robot.

In this chapter I am going to talk about the vision process used to transform the sequence of images into consistent outputs for the tasks. Figure 2.1 describes the mechanism used for both corridor following and ego-motion prediction.

The video camera continually grabs pictures. These pictures are processed in order to extract features. Then a neural network predicts the output using the features of the picture and possibly the difference between the current picture and the last one.

For the corridor following, the requested output is the direction to follow given by the steering angle. The neural network uses only the current image for this part of the project.

For the ego-motion estimation, the requested output is the distance between the two frames whose features were provided to the neural network.

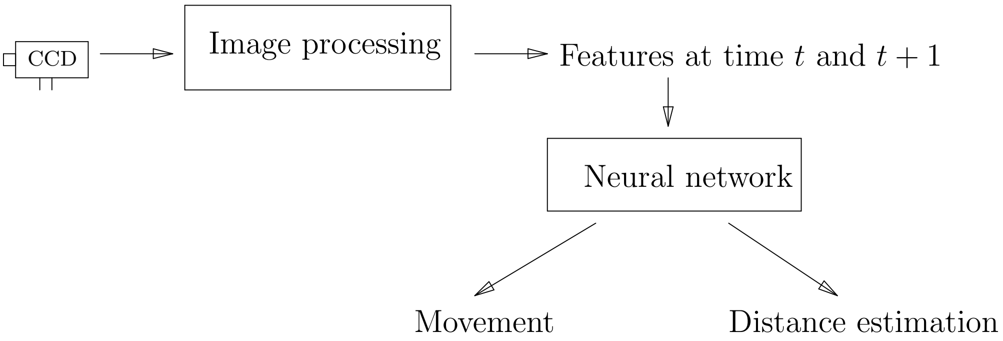


Figure 2.1: Overview of the mechanisms used. The images grabbed are processed to extract features which are used as inputs for a neural network. The neural network then predicts the result, which can be either a steering angle or a distance estimation.

The following sections describes in more detail which features were extracted from the frames and which neural network architectures were used.

2.2 Vision tools

2.2.1 General vision tools

In figure 2.1, we see that the first step is an image processing algorithm that extracts features from the image. This step is a very important one. The accuracy of the results depends on the accuracy of the features extracted.

Indeed, if a neural network is trained with poor features, it provides a poor result. For instance, if we only used the features “the image is bright” and “the image is not bright”, no algorithm can predict the movement the robot has to do. If we only use these features, there is not enough information to decide whether the robot has to turn right or to turn left. So we need an adequate image processing mechanism to extract useful features.

A useful feature has to provide meaningful information. Edges are a feature that is widely used in machine vision and that hold relevant information for most applications. Indeed, ego-motion computation has already been calculated using edge detection, using optical flow [23, 26, 25, 28]. This proves that the edge information is sufficient to compute travelled distances. Here, we expect neural networks to compute this for us, or expect at least to provide an estimation of this computation.

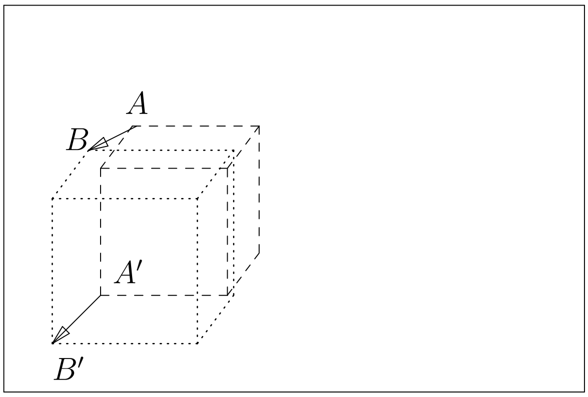


Figure 2.2: Useful features of edge detection. Two successives frames are shown in dashed lines and dotted lines. The distance travelled by the robot can, under certain conditions, be found using vectors $\overrightarrow{AB}$ and $\overrightarrow{A'B'}$

Another strength of edges is the fact that there are a lot of edges where there are a lot of objects. Indeed, borders and textures of objects produce a lot of detected edges. This is useful for corridor following. With these features we can estimate the location of the objects on the images using simple computations. We also expect the neural network to estimate wheel movements using these computations. The next section describes the computations used as preprocessing for the neural network inputs.

Different edge detectors were tried, but the best results were obtained using a Difference of Gaussian edge detector. This edge detector is powerful and cheap. It saves a lot of computation time because each pixel on the edges image is simply a thresholded linear combination of its adjoining pixels. This computation can be done by storing only three lines during the algorithm. So this edge detector is memoryless and efficient.

2.2.2 Vision tools for the corridor following

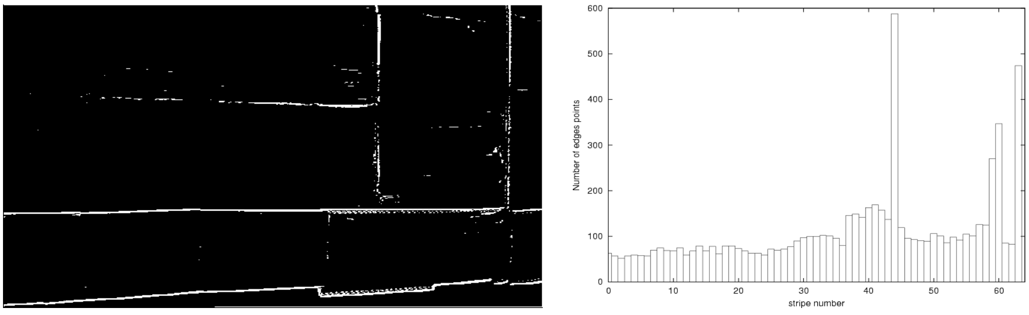
2.2.2.1 Histograms

In order to follow a corridor, a robot has to go forward while avoiding walls. To move forward is not a problem for a robot. The problems arise when it has to avoid walls.

Usually walls generate edges. So if there are a lot of edges detected on the right of the picture, the robot has to turn left, and if there are a lot of edges detected on the left, the robot has to turn right. So we can simply count the number of pixels on the left of the edge-detected image and the number of pixels on the right. We can then compare them and decide which way the robot should turn.

I have implemented this behaviour on *FortyTwo*. It does not work properly because the two walls do not generate the same number of edges. For instance, we can have one wall which is mainly blank and does not generates a lot of edges, and the other wall can be covered by posters or notices. The second wall generates a lot of edges even if the distance between the robot and this wall is the same as the distance between the robot and the other wall. So the robot tries to avoid posters and notices which give good edges instead of the wall itself.

The idea developed in [16] is to split the image into vertical stripes and then count the number of edges detected. This enhances the horizontal location of edges in the image. Indeed, the vertical location of edges does not provide any useful information for a corridor follower. So this feature is more powerful than edge detection only because it contains almost the same quantity of information and it is clearly using less parameters. Figure 2.3 gives us an example of a histogram we can obtain with this method. This histogram is called in this thesis "histogram of the image". This is not a generic name for this kind of method but it makes the method easier to be referred to.



(a) Edge-detected image

(b) Histogram of (a)

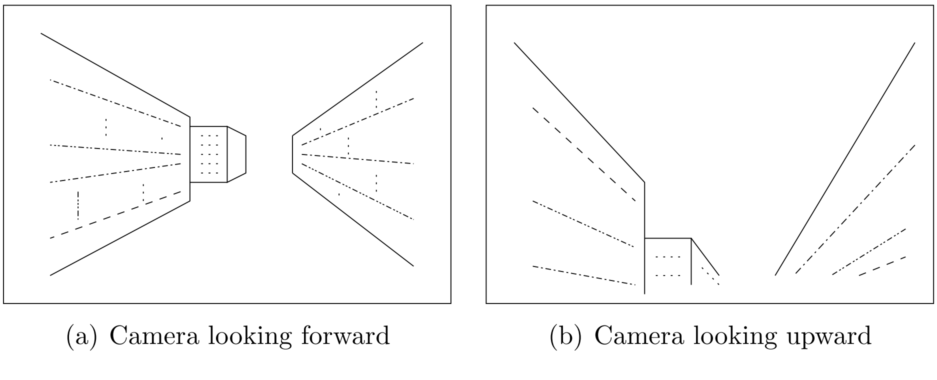
Figure 2.3: Vertical histogram of an image. An edge detection was performed on the image which gives the edges image 2.3(a). This edges image is then split into 64 vertical stripes and figure 2.3(b) gives us the histogram of the number of edges pixels per stripe.

We notice that this method is only viable if the camera is always pointing towards the end of the corridor. Having a camera which is constantly pointing in a fixed direction is only possible with *FortyTwo*, not with *Charm*, because *FortyTwo* has a turret which can rotate independently of its base. In the next subsection, I describe a method which I have used for the case of a camera that turns with the robot.

2.2.2.2 Hough transform

If the camera does not point in a fixed direction, the left wall is not always on the left-hand side of the image, and the right wall is not always on the right-hand side of the image. So in this case, the computation of an histogram of the image is not a solution.

A feature that can be computed is the angle of each of the lines which constitute the edges image, and a fixed direction. This idea has already been implemented in industrial robots such the plant-scale husbandry robot described in [31]. I have taken the vertical direction as the reference direction. It is good to notice that this feature is relevant only if the camera is pointing upwards or downwards, or only the upper or the lower part of the image is used. Indeed, in the edges image, a wall produces lines that form angles of different sign with the reference direction, depending on whether they are in the upper or the lower part of the image. However, if the camera is looking upwards, a wall generates lines that form angles of the same sign, and the other wall generates lines that form angles of the opposite sign. Indeed, all the strong lines seem to intersect at the end of the corridor. Figure 2.4 summarizes this.



(a) Camera looking forward

(b) Camera looking upward

Figure 2.4: Edge-detected images of a corridor. Figure 2.4(a) shows an edges image of a corridor when the camera is looking forwards and figure 2.4(b) shows an edges image of the same kind of corridor when the camera is looking upwards.

The problem is to calculate the angle of each line in the edges image. A common way to do this is to use a Hough transform. The Hough transform is described in [27]. The idea is to use an accumulator array which computes a strength for each line by a voting mechanism. This accumulator array is called Hough space. The dimension of the Hough space is defined by the number of geometric parameters we need to describe a pattern. These parameters describe the position or the scale of the pattern we want to recognize. We must be able to restore the original pattern using these parameters. Here, the pattern is a line, so the dimension of the Hough space is two.

I chose the two parameters as follows. The first one is the distance between the centre of the image and the line, and the second one is the angle between the line and the vertical. So each line in the edge-detected image corresponds to a unique point in the Hough space.

In order to find the lines that are correctly detected in an edge-detected image, each point that forms a line will vote for the corresponding parameters. So in the Hough space, a strong value correspond to a line which is well detected by the edge detector, that is, a solid line in the edge-detected image. We can then select the best lines by selecting the stronger values in the Hough space. This procedure is reliable and widely used in machine vision. Figure 2.5 shows an example of line detection using a Hough transform.

In order to compute the Hough space, we have to change one of the two parameters, for each point in the original image. Knowing that the point we are dealing with is on the line, if we know the distance from the line to the centre, we can find the orientation of the line. It is the same if we know the orientation of the line, we can then find the distance between the line and the centre of the image. The second calculation seems to be simple, and so more efficient on a computer.

Figure 2.6 shows the situation when we try to compute the distance between a line and the centre of the image. (x_0, y_0) denotes the point we are currently considering, and (x_1, y_1) denotes the point located at the centre of the image. D denotes the distance between the line and (x_1, y_1) and $\overrightarrow{V}$ is the normal to this line. The norm of this vector is 1. Finally, θ denotes the angle between the vertical and the line. The aim is to compute D using the other parameters.

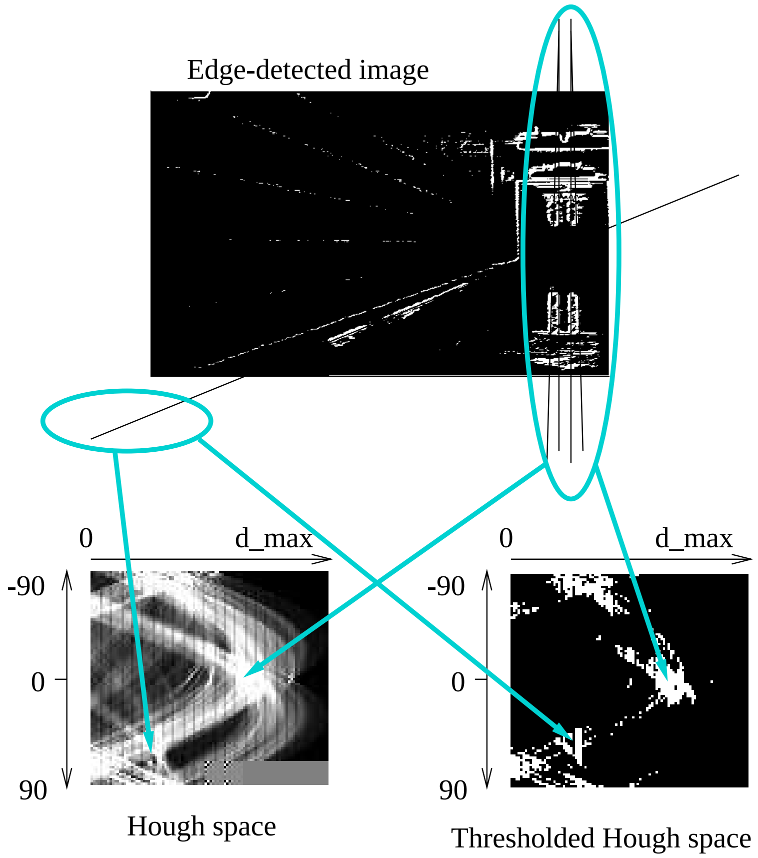


Figure 2.5: Example of the Hough space of an image. The figure shows the result of an edge detection and its corresponding Hough space. An example of the effect of a line and a group of lines on the Hough space is shown. The thresholded Hough space also shows that these lines are strongly detected by the edge detector. d_max is half the length of a diagonal of the image.

Let $\overrightarrow{P}$ denotes the projection of $\overrightarrow{W}$ on $\overrightarrow{V}$. We have:

$$D = \left\|\overrightarrow{P}\right\| = \frac{\left|\overrightarrow{V}.\overrightarrow{W}\right|}{\left|\overrightarrow{V}\right|}$$

Furthermore,

$$\overrightarrow{V} = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

and

$$\overrightarrow{W} = \begin{pmatrix} x_1 - x_0 \\ y_1 - y_0 \end{pmatrix}$$

So

$$D = |\cos(\theta)\,(x_1 - x_0) + \sin(\theta)\,(y_1 - y_0)|$$

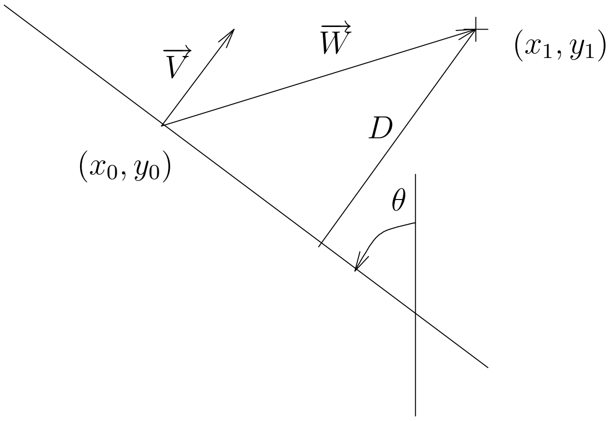


Figure 2.6: Calculation of the distance for the Hough transform. (x_1, y_1) denotes the point located in the middle of the image and (x_0, y_0) denotes the point we are currently using in the algorithm.

So we can see that the distance D can be easily computed if we know the angle θ. Therefore I used the angle θ as the varying parameter to compute the Hough space.

The following algorithm is then used to compute the Hough space (the Hough space is represented by a two dimensional array in this algorithm):

```
compute the edge-detected image using an edge detector.

set all the elements of the array to 0.

for each point in the edge-detected image that is detected
as part of an edge, do:
     for each angle between -90 degrees and 90 degrees, do:
          * compute the distance between the line which goes
            through the point and is directed by the angle,
            and the centre of the image.
          * in the array, increment the value of the element
            labelled by the angle and the computed distance.
     done.
done.
```

```
for each element of the array do:
     * if the value of this element is greater than a fixed
       threshold then the element is set to 1.
     * set the element to 0 otherwise.
done.

return the array.
```

This procedure returns an array which is labelled by angles and distances. We can then use the angle by computing a histogram of the Hough space in the same manner as we computed histograms of images. We can also use the distance information by computing the corresponding histogram. Nevertheless, the distances seem to be less important than the angles. So I used wide stripes to compute the histogram for distances and small stripes to compute the histogram for angles. I then concatenate those two histograms and use it as input to a neural network.

2.2.2.3 Probabilistic Hough transform

Even if the formulas are pretty simple for the computation of the Hough space, it takes too much time to compute the Hough space of an image. If we use the algorithm described in section 2.2.2.2, it takes more than 50 seconds on a 486 PC to perform the Hough transform for a single image. This is too slow for usual operations. So I tried to optimize the computation using both practical and theoretical ideas.

First of all, I decreased the size of the Hough transform so that there are fewer labels and so the part of the algorithm which applies the threshold to the Hough space can be executed quicker. I also precomputed the sine and cosine functions because these functions use a lot of processing power. All the values needed were stored in an array. This array can be used to get values of sine and cosine of angles very quickly.

Another idea which is described in [17] is to slightly change the Hough transform in order to compute the parameters for fewer points. Indeed, it has been proved that computing parameters on only a fraction of the points is enough. The ratio depends of the complexity of the image, but a common ratio is 5%-15%.

This way of implementing the Hough transform is known as the probabilistic Hough transform.

The probabilistic Hough transform was implemented by choosing a ratio of 20% to be sure to keep a decent amount of information. I have not chosen to take points randomly in the image, but take points randomly along lines. I have computed parameters for 20% of the points for each line. This is more practical because the frame grabber provides the image line by line. So this avoids the problem of storing too many lines in memory. We can see in figure 2.7 that the thresholded Hough spaces does not look different for the Hough transform and the probabilistic Hough transform.

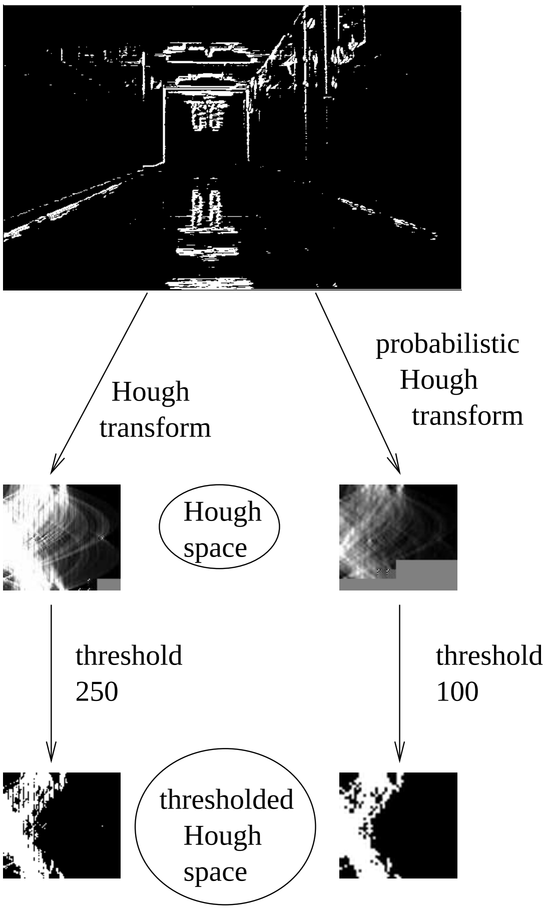


Figure 2.7: Difference between the Hough transform and the probabilistic Hough transform. The figure shows a Hough space and its thresholded version for both standard and probabilistic Hough transforms. Note the different threshold values.

We also notice that the thresholds are different because the peak values are not so strong in the probabilistic Hough space as in the standard Hough space.

Indeed there are fewer points that can vote in the probabilistic Hough transform than in the standard Hough transform.

2.2.3 Vision tools for the egomotion prediction

Egomotion estimation requires the notion of difference between images. A distance cannot be computed with only one picture. We need two pictures to compute it.

Egomotion can be computed, in some circumstances, using optic flow. Optic flow is detailed in a number of articles and books [27, 10, 23, 2]. For this thesis, we only want a neural network to estimate the right computation. So we do not need to compute optic flow explicitly. The only preprocessing used was edge detections and histograms.

In order to compute the egomotion, I used histograms of the two images in both vertical and horizontal directions. In this case we need both horizontal and vertical information because almost all the points of the original image move in both directions. So both kind of histograms provide information. For both images, I concatenated the vertical and horizontal histograms and subtracted the two resulting histograms to provide a set of inputs for the neural network.

The last step is justified by the fact that the neural network takes a linear combination of the inputs and there is no reason to give greater importance to one image compared to another. Furthermore, this operation, in addition to the previous operations, reduces the number of inputs we have to provide to the neural network. This is a worthwhile operation because it reduces the complexity of the neural network and also the complexity of the calculations. Decreasing the number of inputs saves us a lot of computation time, which is a critical resource in our application with a 486 PC.

Histograms do not seem linearly dependent on the distance. Indeed, an object can generate different changes in the image for the same travelled distances, depending on its position in the scene. A object which is close to the camera generates big changes and an object which is far away generates almost no changes in the histograms. Therefore a simple on-line Pattern Associator is not sufficient to learn the relationship between histograms and distances. I have used a multilayer Perceptron instead of a simple Perceptron in order to learn this relationship.

2.3 Off-line multi-layer Perceptron

2.3.1 General description

In mathematical terms, a neural network is simply a function with some parameters called weights. This function can be build using a simple element called a neuron. A neuron can be represented by the figure 2.8.

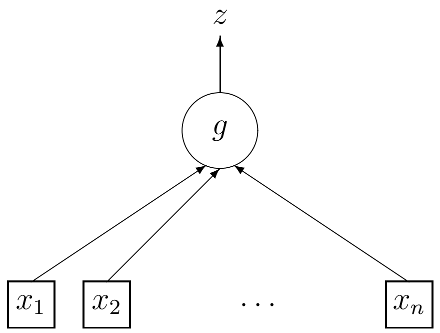


Figure 2.8: Model of a neuron. The output z is calculated using some parameters $\{c_i\}_{i\in[1,p]}$ and a function g : $z = g\left(x_1, x_2, \ldots, x_n, c_1, c_2, \ldots, c_p\right)$.

The function g is composed of a linear combination of inputs which is called potential and an activation function f. The potential ν of a neuron is defined by:

$$\nu = c_0 + \sum_{i=1}^{n} c_i \cdot x_i$$

where c_0 is a constant called bias. The bias can be introduced into the sum by introducing a new input of the neuron whose value is always 1. The bias then becomes the parameter associated with this value. So we can write:

$$\nu = \sum_{i=0}^{n} c_i \cdot x_i$$

where $x_0 = 1$. We can then compute the returned value z using the formula:

$$z = f(\nu) = f\left(\sum_{i=0}^{n} c_i \cdot x_i\right)$$

The activation function used is the hyperbolic tangent function or sigmoid. Other sigmoid functions such as $x \mapsto \frac{1}{1+e^{-x}}$ could be used as activation functions.

These neurons can be combined into a network by providing inputs for neurons using the outputs of other neurons. Such a network is called a neural network. So a neural network can have different number of inputs or outputs, and different architectures.

We are particularly interested in an architecture called the multi-layer Perceptron, which figure 2.9 shows. It is composed of p layers, l_1 to l_p. Each layer is fully connected to the next layer. The output layer has a linear activation function. The other neurons have a sigmoidal activation function.

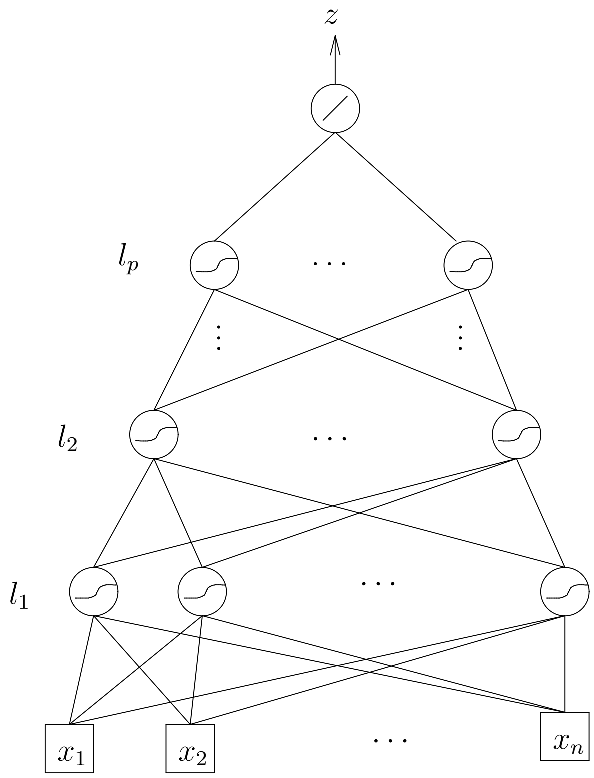


Figure 2.9: Model of a neural network. The output layer uses a linear activation function and the other neurons use a sigmoid activation function. Each layer is fully connected to the next layer.

We also need a way to assess the performance of the learned mapping. The evaluation of the error is done using the sum-of-squares error function over a set T. The set T is a set of pairs. Each pair is composed of an input and the corresponding desired value of the output (target) of the neural network. The sum-of-square function is:

$$E_T(w) = \frac{1}{N}\sum_{i=1}^{N} E_i(w)$$

where w is a vector of weights, N is the number of samples in the set T and

$$E_i(w) = (d_i - z_i(w))^2$$

$z_i(w)$ is the image of the i^{th} input of the set T calculated with the weights w. d_i is the corresponding desired value. $E_i(w)$ is the square of the difference of the result of the neural network for a particular input and the desired value.

This error function has to be minimized with respect to w according to a learning set. The result of this minimization gives us a set of weights which form a trained network. For each regular function, we can find a neural network that can fit this function with an arbitrary precision.

We can assess the performance of the training using a test set which is different from the learning set. The computation of the sum-of-square error over this test set gives us a measure of the performance of the neural network.

2.3.2 Propagation and backpropagation

The gradient of the sum-of-square error function of a multi-layer Perceptron is calculated using the backpropagation algorithm. The propagation algorithm is a preliminary of the backpropagation algorithm.

In order to explain how these two algorithms works, I have to number the neurons from left to right beginning by the layer l_1, then the layer l_2, and so on. Let w_{ij} denote the weight of the link between neuron i and neuron j. Let F_i denote the set of neurons whose output is an input of the neuron i. ν_i is the potential of the neuron i, and z_i is the returned value of the neuron i. Finally, let N_i denote the set of neurons whose set of inputs contains the output of the neuron i.

The propagation algorithm propagates an input vector $\{x_i\}$ through the network. It is used to compute the output of the neural network knowing the inputs and the weights:

```
For $i$ from 1 to the number of neurons, do:

    • For the inputs of the neural network, we define:

        $z_i = \nu_i = x_i$
```

- `For the other neurons of the neural network, we have:`

$$\nu_i = \sum_{j \in F_i} w_{ji} \cdot z_j$$

$$z_i = f_i\left(\nu_i\right)$$

- `For the last neuron of the neural network, we have:`

$$z = z_i$$

`Done.`

This algorithm computes the value of the potential of each neuron. These values are used by the backpropagation algorithm which computes the gradient of the sum-of-square error. It computes $\frac{\partial E_k}{\partial w_{ij}}$ for each weights and for each example, k, in the training set. The algorithm is based on the following factorization:

$$\forall j \in F_i, \frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial \nu_i} \cdot \frac{\partial \nu_i}{\partial w_{ji}} = \frac{\partial E_k}{\partial \nu_i} \cdot z_j$$

We know that d_k denotes the desired output of the neural network if $\{x_i\}$ is the input vector. For the last neuron, we have:

$$\frac{\partial E_k}{\partial \nu_i} = \frac{\partial E_k}{\partial z} \cdot \frac{\partial z}{\partial \nu_i} = -2\left(d_k - z\right) \cdot f_i'(\nu_i)$$

where i is the number of a neuron located on the layer l_p, and z is computed with the propagation algorithm. For the other neurons, we have:

$$\frac{\partial E_k}{\partial \nu_i} = \sum_{j \in N_i} \frac{\partial E_k}{\partial \nu_j} \cdot \frac{\partial \nu_j}{\partial \nu_i}$$

but:

$$\nu_j = \sum_{i \in F_j} w_{ij} \cdot z_i = \sum_{i \in F_j} w_{ij} \cdot f_i(\nu_i)$$

thus:

$$\forall j \in N_i, \frac{\partial \nu_j}{\partial \nu_i} = w_{ij} \cdot f_i'(\nu_i)$$

finally:

$$\frac{\partial E_k}{\partial \nu_i} = f_i'(\nu_i) \sum_{j \in N_i} w_{ij} \cdot \frac{\partial E_k}{\partial \nu_j}$$

These computations can be summarized with the following algorithm :

```
For all couple of inputs/output in the training set, do:

  • Propagate the $k^{th}$ input vector into the network using
    the propagation algorithm.

  • For $i$ from the number of neurons to 1 do:

      – If the neuron $i$ is the last neuron, compute:

            $\frac{\partial E_k}{\partial \nu_i} = -2\,(d_k - z) \cdot f_i'(\nu_i)$

      – Otherwise, compute:

            $\frac{\partial E_k}{\partial \nu_i} = f_i'(\nu_i) \sum_{j \in N_i} w_{ij} \cdot \frac{\partial E_k}{\partial \nu_j}$

    Done.

  • For each weights $w_{ij}$, compute:

            $\forall j \in F_i, \frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial \nu_i} \cdot z_j$

    where $z_j$ has been computed with the propagation algorithm.

Done.
```

Two methods were used for the minimization of the error function. In a first stage, we try to approach a minimum of the error function using a gradient descent. In a second stage, we use a Quasi-Newton algorithm to find the minimum. The Quasi-Newton algorithm is only valid in a neighbourhood of the minimum. It has to be initialized with a point close to the minimum. That is why we have to use a gradient descent before using the Quasi-Newton algorithm. The next section describes these two minimization algorithms.

2.3.3 Learning algorithms

2.3.3.1 Gradient descent

The method of gradient descent is one of the most fundamental procedures for the minimization of a function E of several variables. In order to find a minimum, we begin at random values for the variables, which gives us a point w_0 in the original space. We can then choose another point by moving in a direction d. The new point must have an image that is smaller then the image of the previous point. We then move again in another direction that decrease the value of the function to find another point, and so on.

Using this idea we keep decreasing the value of the function until we found a minimum. For each iteration, the first step is to find the right direction. We have to choose a descent direction. As we know that the opposite of the gradient is the steepest descent direction, we can choose the direction d_k for each point w_k as follows:

$$d_k = -\nabla E\left(w_k\right)$$

A second step, which is harder to find, is the length of step along each direction d_k. If the step is too big, the algorithm can diverge. If it is too small, the algorithm takes a long time to converge. Some choices like a step of $\frac{1}{k \cdot \|\nabla E(w_k)\|}$ are proved to converge [18]. However, these methods are often slow. Here, our goal is only to get quickly close to a minimum. So we can take a step of $\frac{\eta}{\|\nabla E(w_k)\|}$ where η is a constant. With this step, we are also sure that the algorithm will not diverge until a big number of steps because the distance between two points is less than 1.

So the algorithm becomes:

```
Choose a random point $x_0$.
For $k$ from 1 to a fixed number of iterations:
```

$$w_{k+1} = w_k - \eta \frac{\nabla E\left(w_k\right)}{\left\|\nabla E\left(w_k\right)\right\|}$$

2.3.3.2 Quasi-Newton algorithm

As the former algorithm is slow, especially in the neighbourhood of a minimum, we need a more powerful algorithm to find the minimum quickly. The Quasi-Newton algorithm is based on the idea that a function looks like a quadratic function near its minimum. In the case of a quadratic function, we have :

$$w_{minimum} = w_0 - \left(\nabla^2 E(w_0)\right)^{-1} \cdot \nabla E(w_0)$$

so if we use this equation to update the weights, the new weights should be close to the minimum. The minimum can then be reached in a few steps. In the ideal case of a quadratic function, the minimum is found in one iteration.

In practice, the computation of the Hessian of E is very slow, so the convergence speed of this algorithm is not really as good as it is supposed to be. It is usually better to approximate the Hessian matrix in order to find a good descent direction. The BFGS (Broyden, Fletcher, Goldfarb and Shanno) formula was used to approximate the Hessian [1]. The descent direction becomes:

$$d_k = -M_k \cdot \nabla E\left(w_k\right)$$

where M_k is computed using the BFGS formula:

$$M_{k+1} = M_k + \left[1 + \frac{\gamma_k^T \cdot M_k \cdot \gamma_k}{\delta_k^T . \gamma_k}\right] \frac{\delta_k \cdot \delta_k^T}{\delta_k^T . \gamma_k} - \frac{\delta_k \cdot \gamma_k^T \cdot M_k + M_k \cdot \gamma_k \cdot \delta_k^T}{\delta_k^T . \gamma_k}$$

where $\delta_k = w_{k+1} - w_k$ and $\gamma_k = \nabla E\left(w_{k+1}\right) - \nabla E\left(w_k\right)$.

Even if the BFGS formula preserves positive definiteness, the computation of M_k can lead to a matrix which is not positive due to rounding errors. So for each iteration, we must have $\delta_k \cdot \gamma_k > 0$. If it is not the case, we simply initialize M_k to the identity. It is proved that (M_k) tends to $\left(\nabla^2 E(w)\right)^{-1}$ [18]. So the formula tends to produce the same result as the idea formula used for a quadratic curve.

We also need the step for the update as it was the case in the gradient descent. Now this stage is more important because the minimization requires more precision near the minimum. However, the Quasi-Newton algorithm is not very sensitive to the step chosen. Thus, we can use a simple and efficient algorithm to choose this step like the Nash method [18].

These stages combined together lead to the following algorithm:

```
Initialize a tolerance factor $\tau$ between 0 and 1.
Initialize a reduction factor $r$ between 0 and 1.
Initialize the weights $w_0$.
Initialize $k = 0$, $g_0 = \nabla E\left(w_0\right)$, $M_0 = I$, $d_0 = -M_0 \cdot g_0$, and $\mu = 1$.
Do:
   • Step 1:
     If $E(w_k + \mu \cdot d_k) \leq E(w_k) + \mu \cdot \tau \cdot g_k^T \cdot d_k$ accept $\mu$.
     Else compute $\mu = \mu \cdot r$ and go back to step 1.
   • Step 2:
     Compute the new weights $w_{k+1} = w_k + \mu \cdot d_k$.
     Compute the new gradient $g_{k+1} = \nabla E(w_{k+1})$ using
     the back propagation algorithm.
     Compute the new matrix $M_k$ using the BFGS formula.
     Compute $d_{k+1} = -M_{k+1} \cdot g_{k+1}$.
     Initialize $\mu$ to $1$ again for the next iteration,
     and increment $k$.
Until a certain number of iterations is reached or
until the weights are almost not changed
during the iteration ($\|g_k\| \approx 0$).
```

In practice, this algorithm works fine with small networks. Indeed, the dimension of the matrix M_k is the square of the number of weights of the network. So this matrix takes a lot of space in memory. This becomes a problem when we begin to use networks with a few thousand weights. In order to bypass this problem, we used a memoryless version of the Quasi-Newton algorithm [4]. This version is based on an approximation. In the BFGS formula, M_k is replaced by the unit matrix I. We can then update the search direction using the formula:

$$d_{k+1} = -g_{k+1} + A \cdot \delta_k + B \cdot \gamma_k$$

where A and B are two reals defined by:

$$A = -\left(1 + \frac{\gamma_k^T \cdot \gamma_k}{\delta_k^T \cdot \gamma_k}\right) \frac{\delta_k^T \cdot g_{k+1}}{\delta_k^T \cdot \gamma_k} + \frac{\gamma_k^T \cdot g_{k+1}}{\delta_k^T \cdot \gamma_k}$$

and

$$B = \frac{\delta_k^T \cdot g_{k+1}}{\delta_k^T \cdot \gamma_k}$$

As we can see, this version of the Quasi-Newton algorithm only requires to store vectors and no matrix. So if W is the number of weights of the network, the memory storage used is $\mathcal{O}(W)$ instead of $\mathcal{O}(W^2)$.

2.4 On-line Pattern Associator

A Pattern Associator is a neural network with only one layer of neurons. This architecture was only used with one neuron because we only needed one output. The architecture used is shown in figure 2.10:

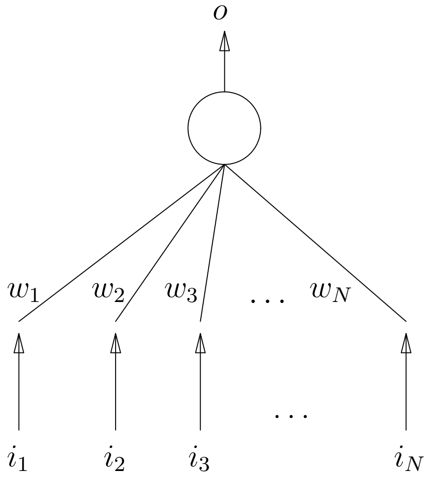


Figure 2.10: Pattern Associator. Each input i_k is weighted by the weight w_k. The activation function of the neuron is the identity function.

The output of the Pattern Associator is computed using the formula:

$$o = \sum_{k=1}^{N} w_k \cdot i_k = \overrightarrow{w} \cdot \overrightarrow{\imath}$$

The aim is still to minimize error between the expected value and the computed value for given inputs. So the weights are updated using the following formula [20, 19]:

$$\overrightarrow{w} \leftarrow \overrightarrow{w} + \eta (t - o) \overrightarrow{\imath}$$

where o is the current output generated by the Pattern Associator, t the corresponding expected value, $\overrightarrow{w}$ the weights at the current iteration, $\overrightarrow{\imath}$ the current inputs and η a constant called learning rate. The learning rate may vary from one iteration to another. Considering the same arguments as the multi-layer Perceptron, the following formula was used:

$$\eta = \frac{1}{\|(t - o)\overrightarrow{\imath}\|}$$

If the training examples are linearly separable, the Pattern Associator converges [19]. Furthermore, another advantage of the Pattern Associator is its low memory consumption. Each learning step, that is each application of the update formula, only requires the current inputs $\overrightarrow{\imath}$ and the current target t. It is not necessary to store all the former training values. This avoids all the memory problems we had with the multi-layer Perceptron. The update formula is also simple. It does not require a lot of computing power. These points are key points in embedded systems. So the Pattern Associator is well suited for robotics, as soon as the training examples are linearly separable.

Chapter 3

Corridor following experiments

3.1 General experimental procedures

The corridor following experiments were done in two distinct parts. The two parts were done with different robots, so the procedures used are different. I describe the robots used in the relevant sections.

Usually, for testing or in regular use, the robot is autonomous and is not controlled by the operator. If the robot has to follow a corridor, we do not want the operator to modify the trajectory. The robot must perform its task by itself, without hitting anything. I therefore implemented a safety procedure based on infrared or sonar, depending which robot was being used. When a obstacle is detected using these sensors, the robot turns until there are no more obstacles, then moves forward for a while and finally restore its original direction by rotating using the opposite angle, so that the direction is the same as before. The general procedure used in normal behaviour is shown in figure 3.1. A neural network is trained by the designer of the robot. This procedure is done only once and produces a trained neural network that can be used each time afterwards. In regular use, each preprocessed image is used as an input to the neural network previously trained. The propagation of these inputs results in a prediction of the right movement to do in that particular situation. The following sections explain all the experimental procedures for each different experiment.

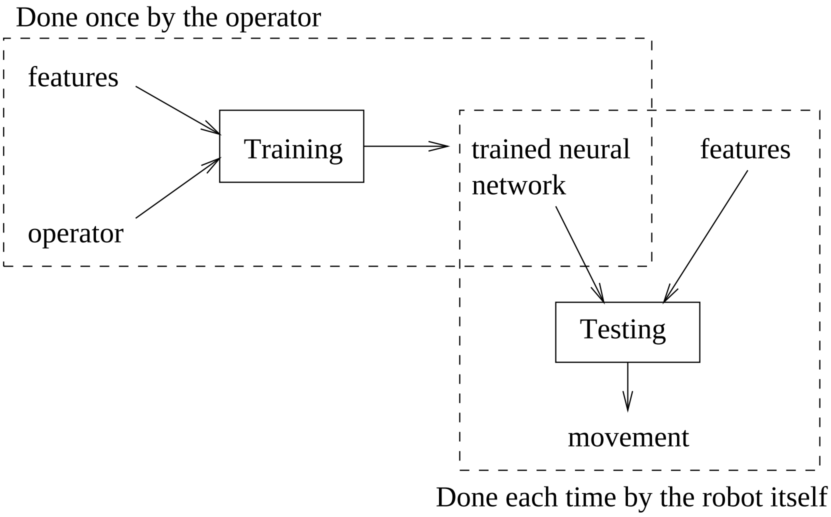


Figure 3.1: General procedures for corridor following. The operator trains a neural network which is used afterwards with a sequence of preprocessed images to control the behaviour of the robot.

3.2 Experiments with *FortyTwo*

3.2.1 *FortyTwo* hardware properties

FortyTwo is a Nomad 200 robot from Nomadic Technologies Inc [21]. It is a wheeled, fully autonomous robot.

FortyTwo has 16 sonars sensors, 16 infrared sensors and 20 bumper sensors. These sensors are arranged in rings around the turret and the base.

FortyTwo has a three servo synchronous drive system powered by two 24 volt batteries. This allows the user to control the speed of the wheels and turn the turret and the base of the robot independently. There are two slots to plug batteries for motor control in parallel. Table 3.1 summarizes the maximum speed of each axes and other physical characteristics of *FortyTwo*.

The turret hosts a 486 PC (66 MHz) with 8 Mb of RAM and a 1Gb hard disk. A separate 24 volt battery powers the processor. We can change the battery by plugging another one in parallel without interrupting the processing. A speech synthesizer card is connected to the PC to provide full speech capability for the robot. A “video blaster” frame grabber is also connected to the PC in order to use a CCD camera. This card has its own memory which is directly mapped into RAM. A CCD monochrome camera is linked to it using a coaxial wire. The camera can provide 640x480 pixels at a frame rate of 25 images per second, which

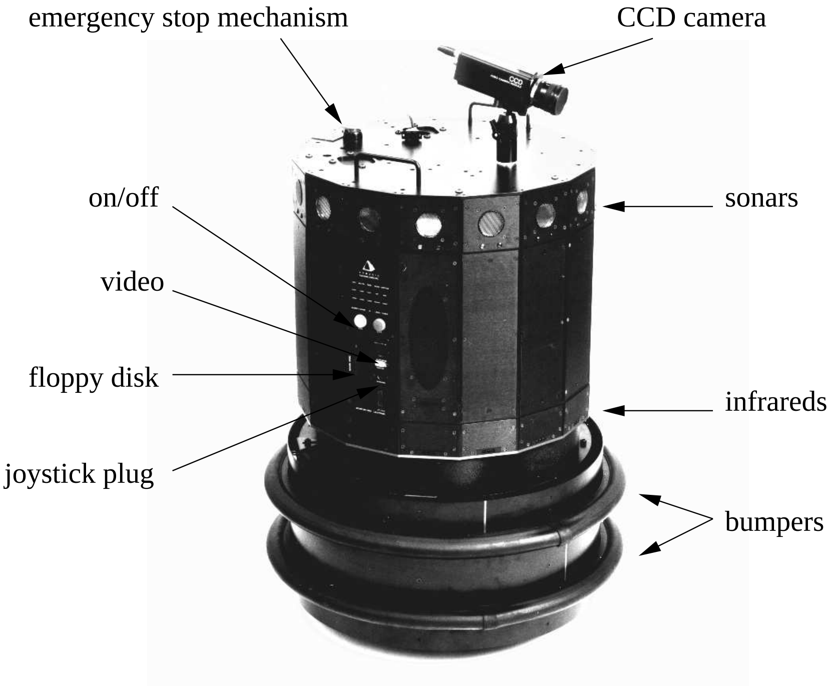


Figure 3.2: *FortyTwo*

is more that the computer can process.

The robot can be controlled remotely via the Internet. Indeed, it is linked to the network with a radio link. This wireless network interface allows the user to run programs even if the robot is in another room or in a corridor. A graphical user interface can also be used to see the state of the sensors and to control the robot with a virtual joystick. The same graphical user interface can be used as a simulator, by placing virtual objects modelled by polygons in the scene.

3.2.2 Description of the experiments

The first experiment was done in the laboratory using a fake corridor made of boxes. The other experiments were in a real corridor. Each image grabbed by the robot is processed as described in figure 3.3.

For all these experiments, the robot was continuously moving forwards using a fixed speed. For each iteration, an image was captured and processed by the robot as described above. The robot then turned its wheels towards the direction computed by the neural network.

Another program was used to capture the position of the robot each time it

Table 3.1: Physical characteristics of *FortyTwo*

diameter	53 *cm*
height	76 *cm*
weight	59 *kg*
payload	23 *kg*
maximum speed	50 *cm/s*
maximum steering speed	60 *degrees/s*
maximum turret speed	90 *degrees/s*
active range of sonars	15 *cm* to 10.60 *m*
accuracy of sonars	1 %
resolution of sonars	2.5 *cm*
active range of infrareds	0 *cm* to 60 *cm*

moved, in order to see how well the robot performed.

The experiment was done in two parts. The first part is the learning part. The experimenter trained the robot with the joystick. In the training stage, the target output was taken from the joystick and the inputs from the processed image. In the testing stage, the inputs of the neural network was taken from the processed image, and the output was computed using the propagation algorithm which is only a weighted sum here.

3.2.3 Results

Figure 3.4 shows the first experiment done with *FortyTwo*. The camera was not correctly oriented during the beginning of the testing. After a few iterations, I oriented it correctly so that it looks towards the end of the corridor. The camera was correctly oriented during the training which is represented by the sparse points in the odometry records shown in figure 3.4.

We see that the robot is able to follow the right direction after a while. The movement is smooth thanks to the way we drive the robot using a constant speed. It is interesting to see that at the beginning of the experiments the robot tries to avoid a box which is in front of it. The robot not know which way to go to avoid the box because the box does not generate a lot of edges when seen from this point of view.

Results of experiments done in the real corridor are shown on figures 3.5(a) and 3.5(b). These figures show that the robot does not take the right direction,

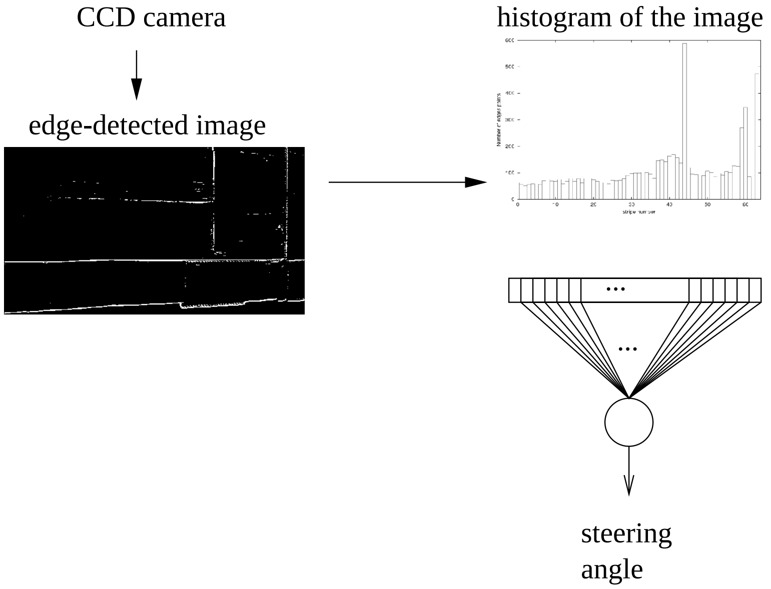


Figure 3.3: Overview of the experiments with *FortyTwo*. Each image is processed by the edge detector in order to extract the histogram of the image. The values given by the histogram are then processed by a Pattern Associator in order to compute the steering angle used for the movement.

but a slightly different one. Indeed, it has not be trained enough to perform its task accurately. Furthermore, the Pattern Associator used did not contain any bias. Adding a bias, that is a constant input, should correct this behaviour.

A change in the hardware used prevented us from doing other experiments with *FortyTwo*. Further experiments in corridor following were done with another robot: *Charm*.

3.3 Experiments with *Charm*

3.3.1 *Charm* hardware properties

Charm is a Scout robot from Nomadic Technologies Inc [14, 22].

Charm is not independent. It has to be controlled by a host computer via a serial link. A Motorola MC68332 main processor is responsible for this communication and a DSP processor is responsible for the motor control. Table 3.2 summarizes its physical characteristics.

Charm is equipped with 16 sonar sensors and 6 independent bumper switches. It has a two wheel differential drive system. The user can provide the speed of

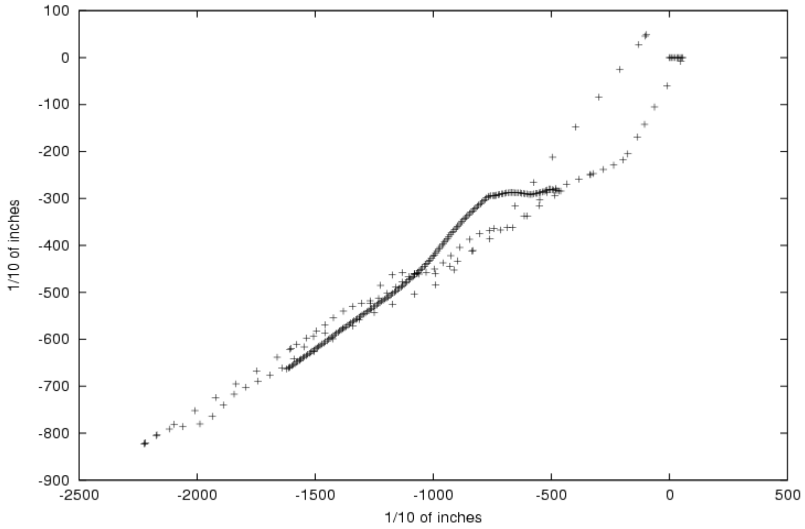


Figure 3.4: Experiment with *FortyTwo* in the laboratory. This figure shows the odometry records during the first experiment. The sparse points represent the training set and the regular spaced points represent the testing. At the beginning of the testing the camera was not looking in the right direction. It was adjusted after a few testing iterations.

each motor independently.

Charm is controlled by a 486 PC running Linux via its serial port. Therefore, the software of the Scout provided by Nomadic Technologies is mostly compatible with the software of the Nomad 200 [21, 22]. This allowed us to change the robot during the project and to run the same code with only a few changes in it. The same graphical user interface as *FortyTwo*'s one can be used with *Charm*.

3.3.2 Difference in the hardware between *FortyTwo* and *Charm* important for the corridor following

The first difference comes from the way to give orders to the motors of each robot. The fact that *Charm* uses the serial port to receive orders does not allow us to control it from different programs. This restriction can be bypassed by using threads in the program that controls the robot.

The different drive systems have also a great impact on the way of programming each robot. As we can see on figures 3.7(a) and 3.7(b), *FortyTwo* is able to turn its turret independently of the position and speed of its other axes. Furthermore, in our experiments, the camera was placed on the top of the turret. So

Table 3.2: Physical characteristics of *Charm*

diameter	41 cm
height	35 cm
weight	23 kg
payload	5 kg
maximum speed	1.0 m/s
maximum acceleration	2 m/s^2
encoder resolution:	
translation	756 $counts/cm$
rotation	230 $counts/degree$

we were able to fix the direction of the camera.

With *Charm*, it is not possible to fix the direction of the camera anymore. Indeed, only the speed of the two wheels can be controlled. So all the parts of the robot are turning when we have to turn the robot, including the camera.

Another points with differs is the way we moved with *FortyTwo*. It is not possible to specify a constant speed without a lot of computation. The fact that *FortyTwo* had different axes for the steering angle and for the speed of the wheels makes it easier to drive. For *Charm*, it is better to adopt another way of control.

3.3.3 Modified corridor following

3.3.3.1 Description

In order to learn a correct mapping with a Pattern Associator, it is better to have a linearly separable mapping to learn. It is also better if we use a bias to give more mapping possibilities to the Pattern Associator. As the camera is not looking at a fixed direction anymore, we need to select better features. The histogram is not a good feature anymore because it does not contain a lot of information about the direction of the camera. Trying to find the angles between lines and a fixed direction in the picture is a much better idea. These angles seem to change in the same way as the angle between the camera and the direction of the corridor does. That is why a Hough transform was used to process images with that kind of motor control.

Figure 3.8 shows the process used to find the steering angle to use. Each image grabbed by the robot is preprocessed by the edge detector in order to compute the Hough space of the image, using the probabilistic Hough transform. Then, both vertical and horizontal histograms of the Hough space are computed. As positions of the lines in the image is less important than angles in our case, the thickness of the stripes are different for the vertical and the horizontal histogram. 45 stripes were used for the histogram which correspond to the angles, and only 5 stripes were used for the histogram which correspond to the distance of the detected lines to the center of the image.

Furthermore, I used the joystick to teach the robot. In the training sequence, each time an image is grabbed, the steering is checked in order to get a corresponding steering angle. As the joystick overrides the program for the control of the motors, the steering angle can be changed via the joystick. During the testing stage, the joystick is not plugged in anymore, and the steering angle is controlled by the program.

3.3.3.2 Results

The results of this experiment were bad. The robot simply went toward the walls instead of following the direction of the corridor. Even when a multi-layer Perceptron was used to learn the mapping between processed images and steering angles, it did not improve anything. The neural network can learn the examples from the training set, but is unable to generalize. The error computed on a test set kept increasing as the robot was trained.

The problem with this experiment is the way of teaching the robot. Using the joystick to teach the robot introduces some inconsistencies in the training set. Indeed, it is very hard for a human to know exactly the right speed to give to the steering. If the experimenter turns the robot too slowly, the robot tends not to turn at all in the testing stage. If the experimenter turns the robot too quickly, the robot can turn roughly 180° between two frames and look in the wrong direction for the next frame. Unfortunately, the computation of the Hough transform on a 486 PC is so slow that we were always in the second case, even if we turned the robot slowly.

So the way of training the robot had to be changed. The geometry of the robot does not allow humans to estimate correctly the steering needed to follow a wall. Indeed, the camera is placed near the ground and is looking upwards. It is not the usual way of following a corridor for a human. So we opt for an automatic learning, that is, a program which taught the robot how to follow a corridor. The robot should then be able to generalize what it has learned. There are advantages and drawbacks to this method. The main advantage is that the robot is able to learn without the intervention of the human, so the learning stage produces a consistent set. Another advantage is that the robot can use other landmarks, not those that the designer used. A drawback is that the programmer has to find a way of achieving automatic learning. So he has to program a corridor follower.

3.3.4 Corridor following without learning

The corridor follower programmed is based on the vision of the flies described in [13]. The sensitivity of the vision is different for different area in the space. It is the same for humans. If we are looking in front of us, the perception of an object is less difficult if it is in front of us than next to us.

So the idea is to build a sensitivity function that acts as a filter and allows good features to be correctly perceived and bad features to be difficult to perceive. This function is shown in figure 3.9.

The values computed for the histogram which corresponds to the angles of the Hough space are now multiplied by a perception factor. As we can see, angles near 45° and $-45°$ are well detected. On the contrary, angles near 0° and 90° are not detected at all. Indeed this sensitivity function has been designed to rest on diagonal lines in the image which correspond to strong edges of the ceiling. The sensitivity function changes its sign around 0° because if we detect a strong line at 45°, we want to turn right, but if we detect a line at $-45°$ we want to turn left, that is we want to steer using a negative angle.

After applying this filter to the angle histogram of the Hough space, we simply sum the results to find the steering angle we have to use.

As we do not use the joystick anymore for the algorithm, the program can control the motors each time. So we can stop the robot for each processed frame. This avoid blur in the image, and also assure us that the computed steering angle will be used in a same position in the corridor as the inputs were taken.

During tests in a real corridor this algorithm performed very well providing that the robot was correctly positioned at the beginning of the test. If the robot is looking at a wall at the beginning, the robot cannot avoid it because it cannot detect it. But in normal use, the algorithm seemed to perform correctly enough to be used as a teacher for the automatic corridor learning.

A drawback that can be signalled is that if for one reason or another the landmark we used is hidden, the robot cannot choose the right steering angle. For instance, if someone is standing in front of the robot far enough not to be detected by the safety procedure and close enough to hide strong edges of the ceiling, then the robot is lost.

3.3.5 Corridor following with automatic learning

3.3.5.1 Description

The experimental procedure is now quite similar to the experiment described in section 3.3.3.1. The difference is that, instead of using the joystick to teach the robot how to steer, the algorithm described above was used.

So in the learning stage, the robot stops for each frame. Then the corridor following algorithm which does not implement any learning is used to find a good steering angle. The Pattern Associator used in section 3.3.3.1 is then trained with these inputs and steering angle. The robot turns using this computed steering angle and then moves forward for a while before stopping in order to process the next frame.

No human intervention is required during the training. This can be very useful if the training stage takes a long time. Here, only a few tens of metre were required to teach the Pattern Associator. This teaching was done in several times because the robot was attached to its host computer with a wire that was shorter than seven metres. With a longer wire, we could have done the training in one step.

The testing stage is the same as it was in section 3.3.3.1.

3.3.5.2 Results

We expected that the weights of the Pattern Associator would match the perception function that we designed or at least equivalent weights which produces the same output for each inputs. Figure 3.10 shows the weights corresponding to angles in the Hough space that were learned by the Pattern Associator in this experiment.

The weights almost match the perception function. The difference comes from horizontal and vertical detected lines. These lines correspond to the lights in the corridor that are strongly detected by the edge detector because the lights saturate the pixels of the camera.

We can see that the Pattern Associator used an additional landmark to estimate the steering angle. This brings robustness to the algorithm. Indeed, if a person hides the edges of the ceiling, the Pattern Associator can still estimate a correct angle using the lights as landmarks. Of course, the robot will turn less quickly, but it will at least turn in the right direction. If the lights are hidden then the edges of the ceiling can still be used as landmarks.

The fact that the edges of the ceiling and the edges of the lights are usually distinct and far from each other in the image makes it harder for a single person to hide both features. So the learned relationship between images and steering angle is robust and can be used in real environments.

In order to assess the performance of this corridor following algorithm, several tests were made in different corridors. For each test, we used similar computation as in [20] and [8] to correct the odometry and then compute a correct performance measure based on the same characteristics of each trajectories. Figures 3.11(a) and 3.11(b) show an example of corrected odometry. The first point should be the same for each trajectories and the mean direction is also the same. This transformation is achieved by a combination of a translation and a rotation around the first point.

This transformation allows us to compute the mean distance $\overline{d}$ between the trajectory and the mean direction. This is a measure of performance. If this measure gives a big value, then the trajectory is far away from its mean direction. That means that it is more chaotic than a trajectory which small value of the measure. Indeed in the ideal case, the robot follows the corridor in a straight line, and so the measure gives us the value zero.

Figures 3.12(a), 3.12(b) and 3.12(c) show the results of the test of this algorithm in three different corridors. With the measure of performance we have chosen, we have $\overline{d_a} = 0.54\ in = 1.37\ cm$, $\overline{d_b} = 1.96\ in = 4.98\ cm$ and $\overline{d_c} = 0.66\ in = 1.68\ cm$. So we can see that there is not a big difference between the trajectories and their mean directions. In practice, the robot seems to follow a straight line by going slightly on the right, then slightly on the left, then slightly on the right, and so on.

We can note that the training was done in the first corridor, so the performance of the first test is less realistic than the performance of the other. Indeed the learning stage could have caught the basic configuration of the corridor, that is the location of the doors and posters and the test could then be more successful in this corridor than the others. In fact, it is slightly the case, as we can see in the results of respective performances.

In the second corridor, we can notice that one of the trajectories ends before the end, and turns suddenly. This behaviour was caused by the safety procedure which caught a door frame with the sonars. It should not have caught this door frame because it was far enough, so it is certainly a freak sonar value as it happens sometimes. Anyway this behaviour is not a failure in the corridor following algorithm.

The algorithm used did not fail during the tests. However the robot was always correctly positioned in the corridor, looking at the end of the corridor. Some tests were done to estimate the robustness to the angle between the direction of the corridor and the original direction of the robot. The table 3.3 shows the maximum angle we can use to place the robot at the beginning if we want it to perform its task.

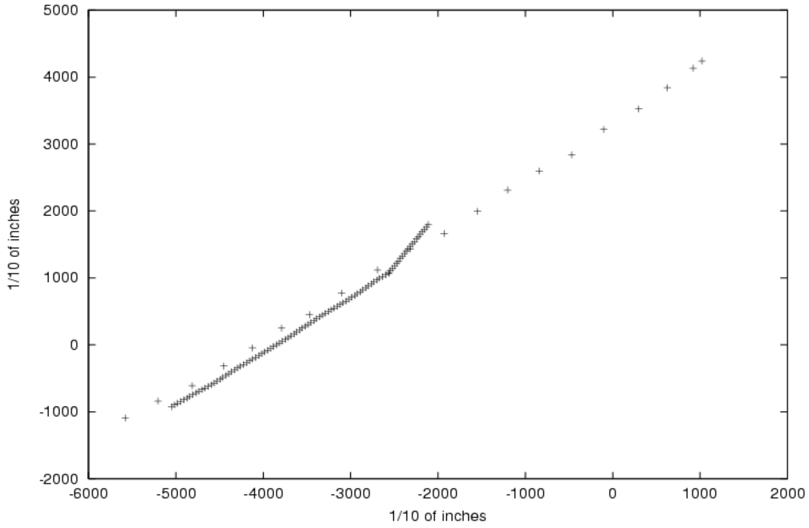
Table 3.3: Maximum original angles between the robot and the corridor.

	left	right
Corridor one	$-11°$	33°
Corridor two	$-11°$	33°
Corridor three	$-11°$	22°

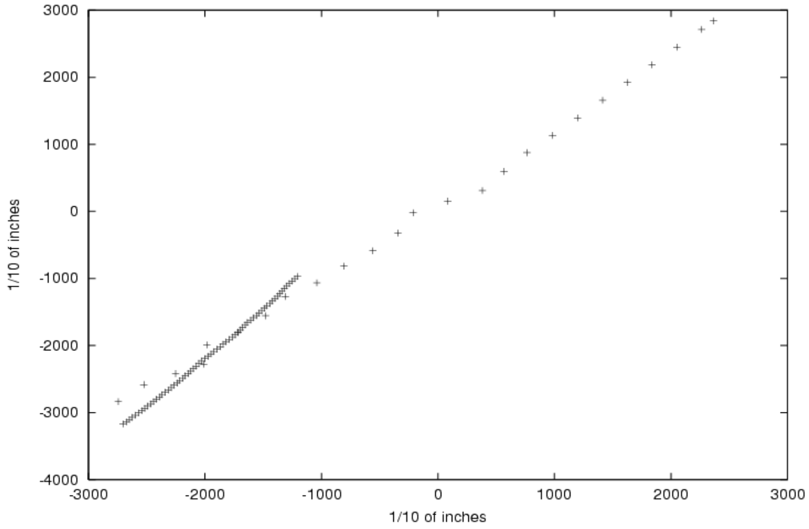
In order to find the figures of table 3.3, the robot has been tested by increasing the original angle it did with the corridor. 11° has been added each time, and when the robot failed, a second test was made with the same angle to be sure that the robot usually fails with this original angle formed with the direction of the corridor. So we can see that the allowed range of angles is not very wide but sufficient enough to dispose the robot without accuracy at the beginning of a test or in normal use. We could not expect a better accuracy because the program that learned the robot how to follow a corridor what already not really good for big angles.

We can also see that the robot performs better if it is oriented toward the right than towards the left at the beginning. This could seem strange because there is no differentiation between the left and the right in the algorithm. Indeed, this difference comes from the layout of the first corridor where the learning took place. There was a notice board near the start position on the left hand side.

Because of this notice board, a big shadow projected on the wall prevented the robot from learning correctly how to avoid a wall when facing it. On the contrary, the right wall was correctly illuminated. So the robot could correctly learned how to avoid a right wall when facing it.



(a) Experiment one



(b) Experiment two

Figure 3.5: Experiments with *FortyTwo* in the corridor. The sparse points represent the training of the robot and the regular spaced points correspond to the testing.

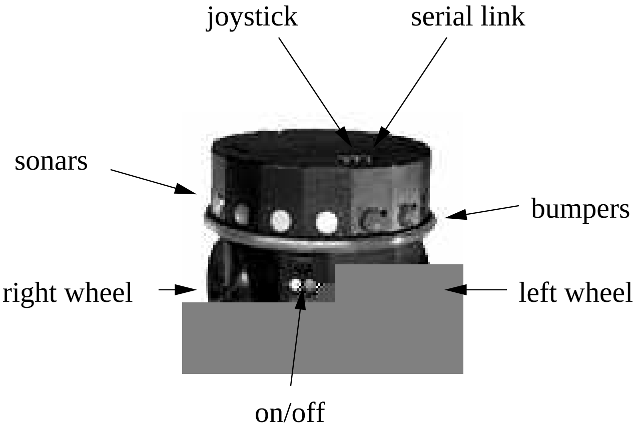
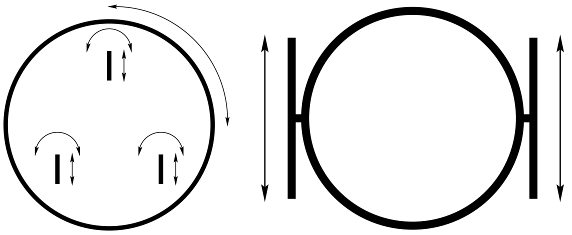


Figure 3.6: *Charm*



(a) Drive system of *FortyTwo* (b) Drive system of *Charm*

Figure 3.7: Drive systems. *FortyTwo* has three independent axes, one controlling the orientation of the three wheels, one controlling the speed of the three wheels and the last one controlling the orientation of the turret. *Charm* has two independent axes, one controlling the speed of the left wheel and the other one controlling the speed of the right wheel.

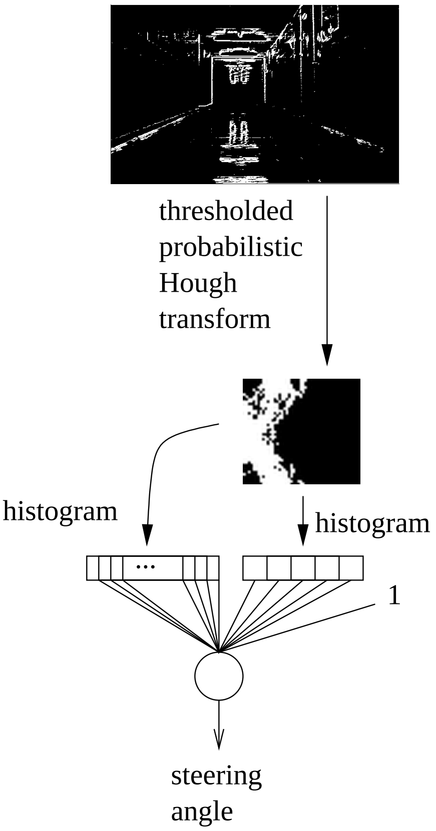


Figure 3.8: Overview of the corridor following with *Charm*. The images are preprocessed by the edge detector and then the probabilistic Hough transform procedure. The vertical and horizontal histograms of the Hough space is then processed by a Pattern Associator in order to return the steering angle.

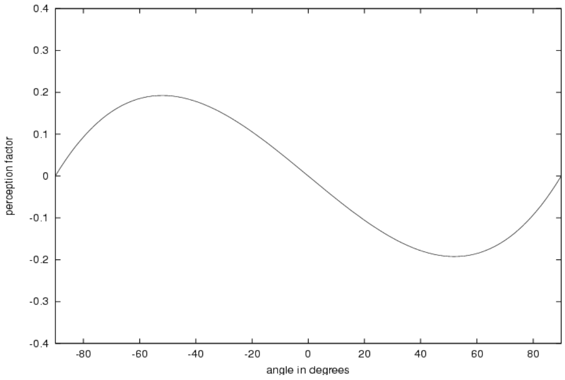


Figure 3.9: Sensitivity function for corridor following. For each angle the value computed for the histogram is multiplied by the corresponding perception factor.

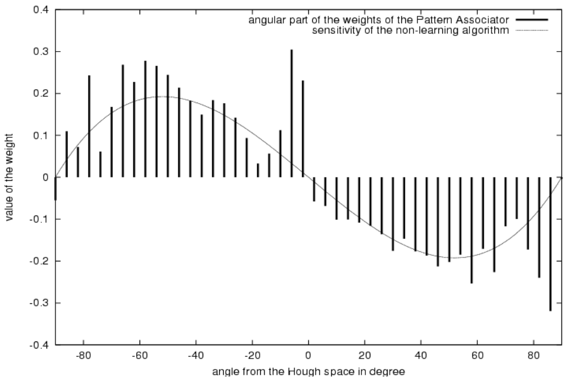
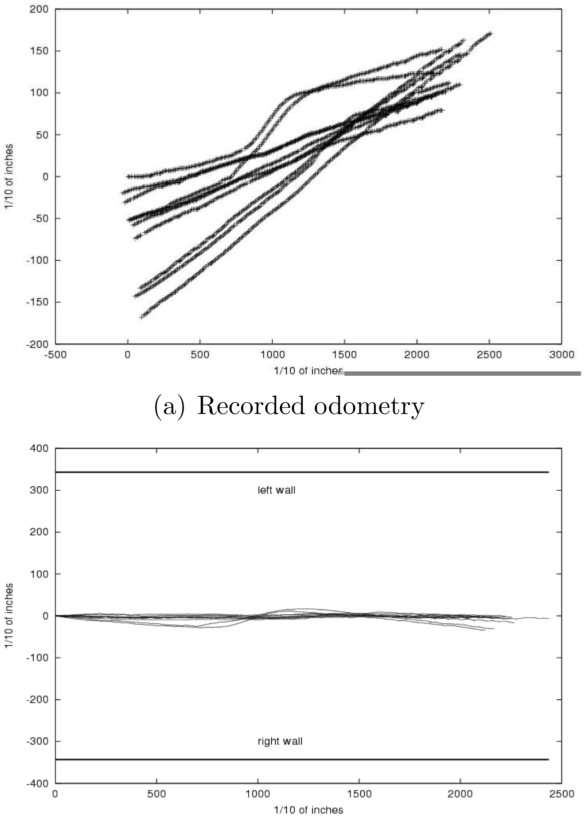


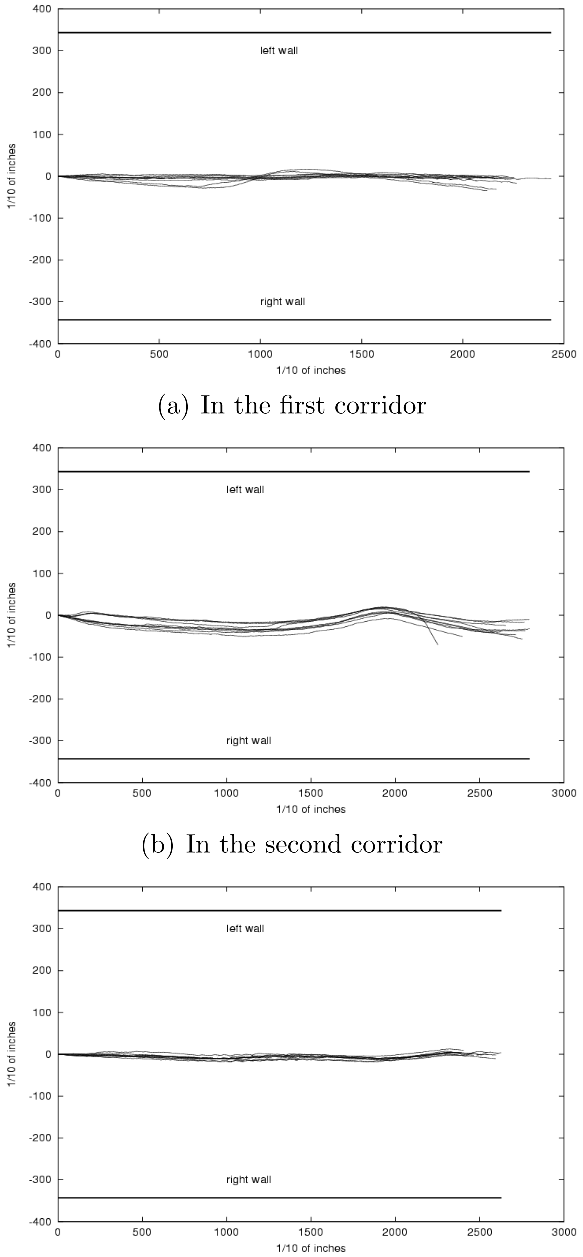
Figure 3.10: Weights learned by the Pattern Associator. The weights match the perception function of figure 3.9 except for angles near 0° and 90°. The detected lines for these angles correspond to the lights of the corridor.



(a) Recorded odometry

(b) Corrected odometry

Figure 3.11: Correction of the odometry drifts. The raw values are translated and rotated to have the same starting point and the same mean direction.



(a) In the first corridor

(b) In the second corridor

(c) In the third corridor

Figure 3.12: Results of corrected odometry for all the experiments. The training was done in the first corridor.

Chapter 4

Egomotion prediction experiments

4.1 Experimental procedure

All the previous experiments in the corridor were using a single frame to control the robot. When dealing with odometry, a single frame is not sufficient. We have to use two frames and compute the distance between the two corresponding locations in the room. All the experiments described in this chapter were done in the robotics laboratory with boxes, tables and computers around.

The idea of automatic learning was applied once more. In the beginning, the robot took a picture of the room. Then it moved forward for a random amount of distance and took a second picture of the room. The distance between the two locations was recorded with the internal odometry system. The robot then moved for a random amount of distance again, took a picture, etc.

Figure 4.1 shows the processus which was used to preprocess the two different pictures. Each picture was preprocessed by the Difference of Gaussian edge detector in order to extract both vertical and horizontal histograms. The histograms were then concatenated. This gives us two vectors, one per original picture. A possible solution is to concatenate these two vectors to form the input vector used by the multi-layer Perceptron. But there is no reason to give bigger importance to one picture compared to the other. Indeed, if we swap the two pictures, we should end up with the opposite value. So if we concatenate the two vectors, the learned weights corresponding to each vector should be of opposite values or at least, equivalent. Therefore, it makes sense to subtract the two vectors. The resulting vector will be the input of the multi-layer Perceptron.

The neural network used this time is an off-line multi-layer Perceptron. This means that we have to store each pair of training values composed by a vector and a distance measured by the internal odometry. In consequence, the training of the neural network is done after the data acquisition. It is possible to train the neural network on another machine. As the robot hosts a very slow 486 PC, we did the training on another computer: a Pentium® III, 500 MHz. Even on such a machine the computation is slow, so a small number of inputs was used due to big stripes. In the experiments, the width of the stripes was 20 pixels for both horizontal and vertical histograms.

Some experiments were done using a uniform probability $\mathcal{U}([0, 100])$ from which random travelled distances were extracted. The distances were chosen between 0 and 10 inches. Other experiments were using a gaussian distribution $\mathcal{N}(50, 20)$. The distances were chosen with a average of 5 inches and a standard deviation of 2 inches, in order to test the algorithm and to see how it performed on a restricted window only. The algorithms which computed such distributions were taken from [11].

4.2 Experimental results

Figure 4.2 shows that the multi-layer Perceptron was able to map a relationship between the pairs in the learning set, that is, between vectors which represents preprocessed images and distances. Indeed, the error computed using a test set decreased during the minimization, so the multi-layer Perceptron performs better and better during the minimization. The parameters used for this learning can be found in table 4.1.

The ideal result of the mapping of the multi-layer Perceptron is the correct distance for each test point. In order to assess our result, we want to see how far away we are from the ideal mapping by studying the correlation between the real output (measures from the odometry) and the computed output (estimations from the multi-layer Perceptron). As the relationship should be linear, we have to compute the correlation coefficient between the real and the computed outputs for each test. Figure 4.3 shows the correlation between estimated and real distances that we computed from a testing set. The linear correlation coefficient is 0.8. This coefficient shows that a relationship exists between the real and the estimated

Table 4.1: Parameters used for the learning of egomotion

multi-layer Perceptron:	
number of layers	4
number of neurons in the input layer	53
number of neurons in layer 2	25
number of neurons in layer 3	10
number of neurons in output layer	1
Gradient descent:	
number of iterations	60
value of η	0.01
Quasi-Newton minimization:	
number of iterations	940
value of τ	10^{-4}
value of r	0.3

output, but this relationship is not perfect.

We can also judge the results by plotting the real and the estimated distances for each sample. This is done in figure 4.4. We can also plot the relative error for each sample. Figure 4.5 shows the relative error of the samples corresponding to the samples shown in figure 4.4.

The mean of the relative errors of random distances taken from an uniform distribution is approximately 0.4, and the standard deviation is greater than 10. Consequently, the standard deviation suggests that the result cannot be trusted. However, we can note that this bad standard deviation is due to some extreme values which look more like isolated cases than general cases. In fact we can see that these isolated cases correspond to test samples with a small distance measured by the odometry system. Ignoring these particular cases, the results are better. Indeed, the mean and the standard deviation will be smaller. Of course, the mean will not plummet too much, but we cannot expect the relative error to be below 10% which is the accuracy of the odometry system used to teach the robot how to estimate distances.

An idea that was worth trying is to eliminate small distances during the tests by using a normal distribution centered on the same mean that the original uniform distribution used. This change gives us a windowed testing. Figures 4.6(a), 4.6(b) and 4.6(c) show that the results are not improved with this distribution. The correlation coefficient is now 0.7 instead of 0.8 above. So the idea of windowed testing does not bring any better results.

However, robots are usually required to navigate for a long time and therefore they travel a long distance. So when we want to design a navigation system, we do not really want to worry about small distances. Long distances are definitely more interesting because the internal odometry of a robot makes bigger mistakes after a long distance than after a small distance. Indeed, errors done are often cumulative because of the nature of the contact between the robot and the ground. Furthermore, the friction coefficient between the robot and the ground changes according to the location. For instance, the friction coefficient was not the same in the laboratory and in the corridors.

In these two cases, visual odometry can be more powerful than the internal odometry of the robot based on wheel encoders. Indeed, visual odometry uses global landmarks. So the accuracy of the measures is the same everywhere and the error is often centered around zero. This was the case for the aforementioned experiments, as we can see on figure 4.7.

As we can see, the error function can be modeled by a gaussian centered around zero. Besides, we know that a sum of gaussian distributions, sharing the same mean and standard deviation, is stable by addition. In particular if the mean of those distribution is zero, then the mean of the sum of those very n distributions remains zero and the standard deviation is $\sqrt{n}$ times the standard deviation of any one of the gaussian distributions contributing in the sum. For instance, if we take 80 distances that come all from a gaussian distribution $\mathcal{N}(0, 15)$ the sum of all those distances comes from a gaussian distribution $\mathcal{N}\left(0, 15\sqrt{80}\right)$.

The estimation of a longer distance was computed using path integration. Consecutive distance estimations were done and added. The resulting estimation of the total travelled distance is shown on figure 4.8. The total travelled distance during the experiments was approximately 10 m. Figure 4.9 shows that the corresponding relative error is less than the previous relative errors. The result is even better than expected. We have a mean of 3.9% of relative error with a standard deviation of 3.3%. So the distance measured by the multi-layer Perceptron is better than the distance given by the odometry in more than 90% of cases. This confirms that visual odometry is better than internal odometry based on wheels for long distances.

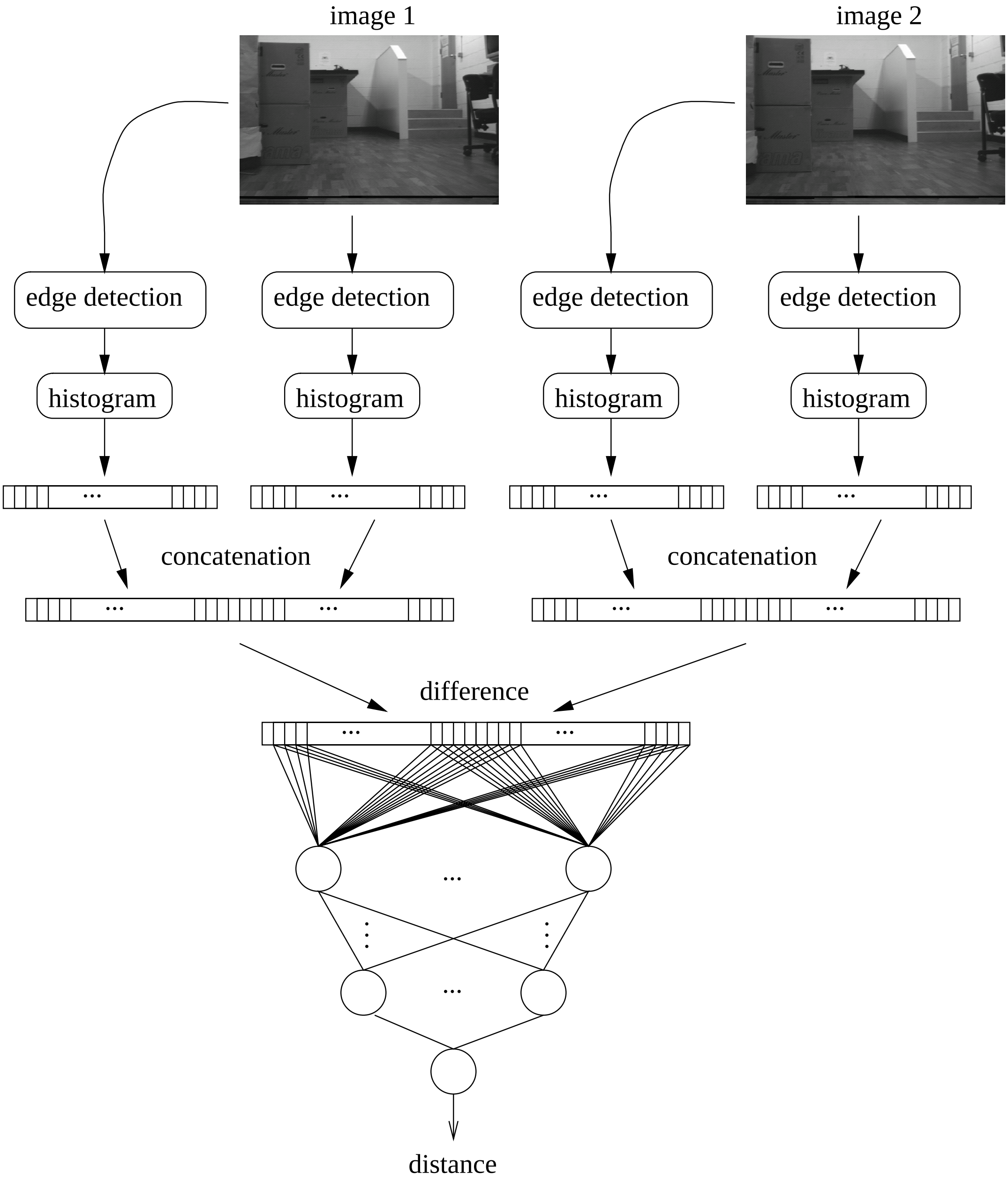


Figure 4.1: Processus used to extract a distance from two images. The concatenation of vertical and horizontal histograms is computed for each image. The difference of the two resulting vectors is the input of a multi-layer Perceptron.

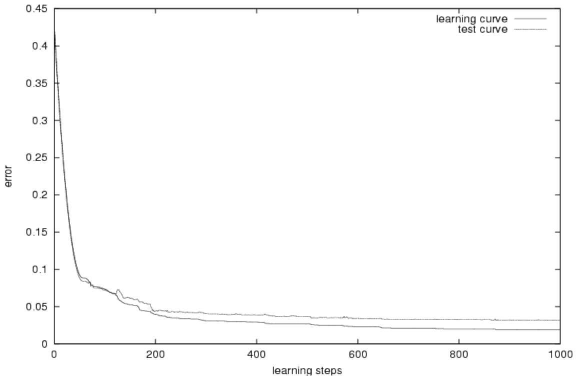


Figure 4.2: Egomotion learning curve. The graph shows the error of prediction of a multi-layer Perceptron the weights of which correspond to each points found by the minimization algorithm. Errors made on the learning set and a test set are plotted for each steps.

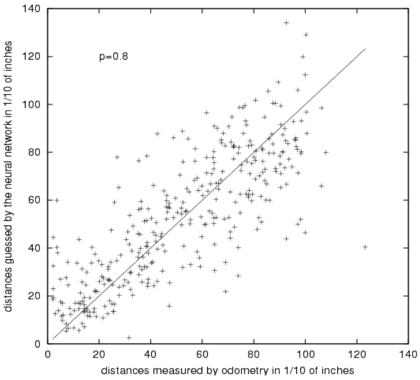


Figure 4.3: Correlation using an uniform distribution. The graph shows the correlation between the real distances given by the odometry system and the computed distances given by the multi-layer Perceptron.

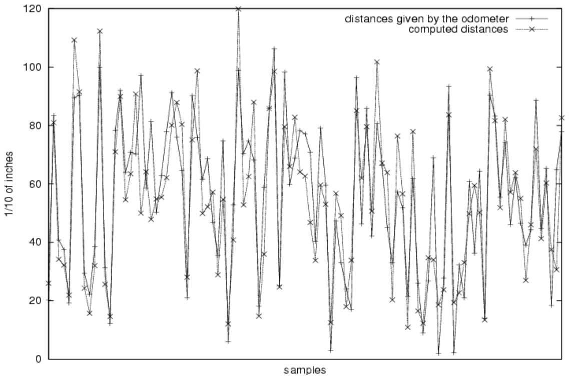


Figure 4.4: Differences between the real and the computed distances. For each sample, the graph shows the real distance given by the odometry system and the estimated distance given by the multi-layer Perceptron.

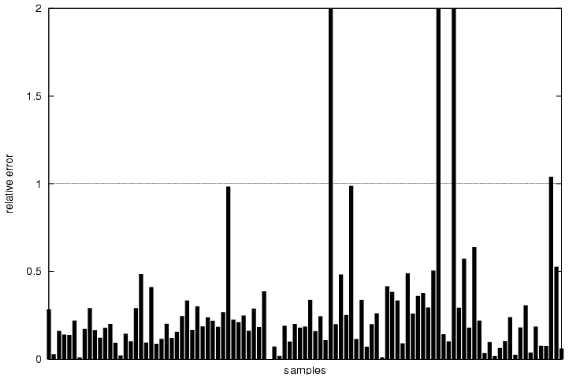
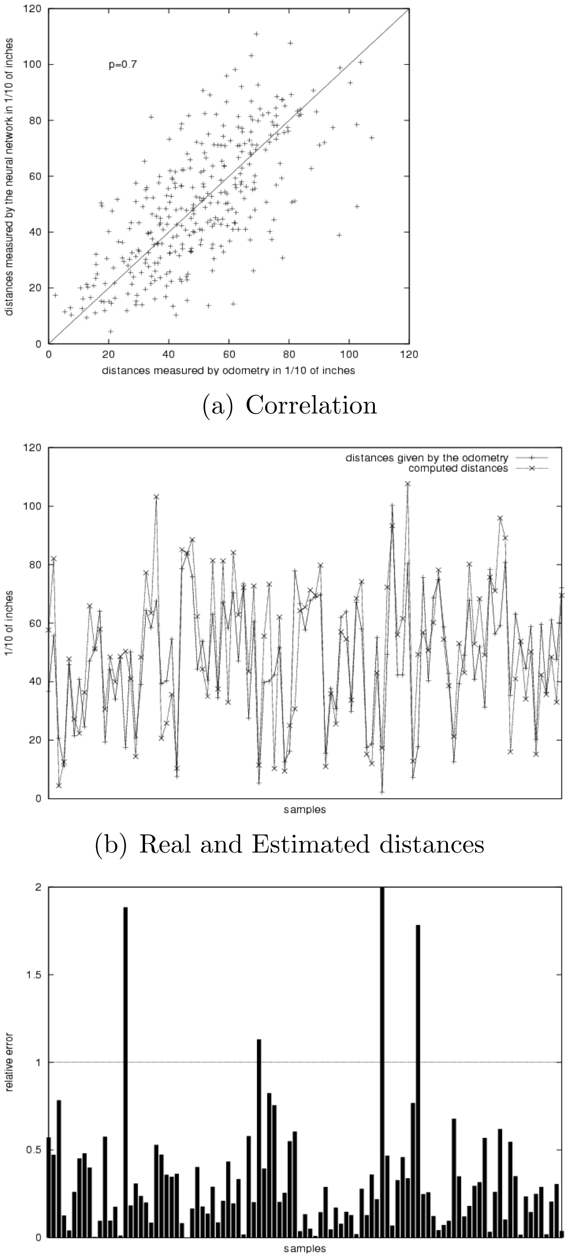


Figure 4.5: Relative errors between the real and the computed distances. Note that the big values of the relative errors correspond to the small travelled distances in figure 4.4.



(a) Correlation

(b) Real and Estimated distances

(c) Relative errors

Figure 4.6: Results using a normal distribution. Big values of the relative error still correspond to the small values of the real corresponding distance. The results are not better than above.

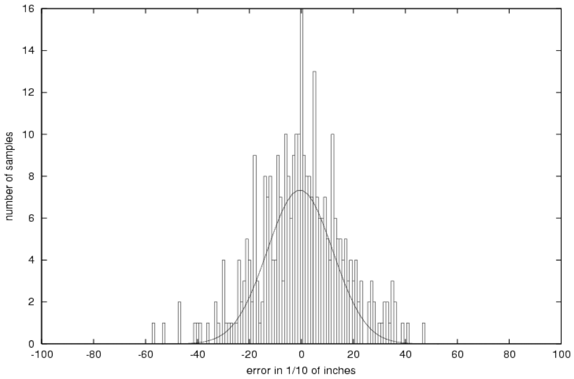


Figure 4.7: Distribution of the error during the tests. The error done by the multi-layer Perceptron seems to follow a gaussian distribution centered around zero.

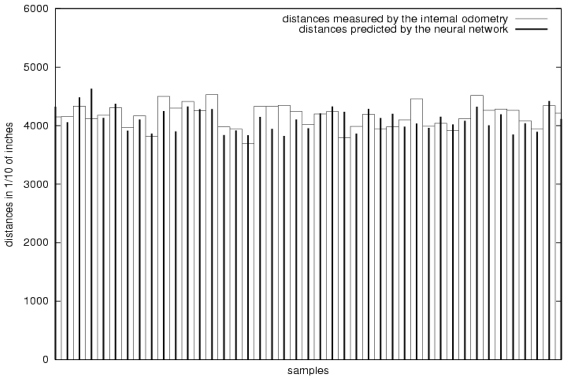


Figure 4.8: Estimation of a long distance. The graph shows the real distances travelled measured by the odometry and the distances estimated by the multi-layer Perceptron.

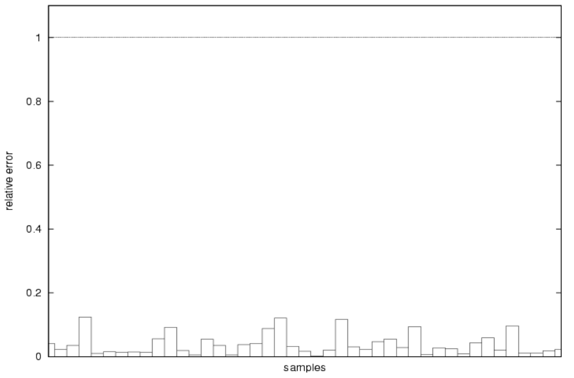


Figure 4.9: Relative error during a long distance. The relative error is less than 0.1 in more than 90% of cases.

Chapter 5

Summary and conclusions

5.1 Summary

In order to achieve our goals, various tools were used. These tools can be divided into different categories.

Programs in a mobile robot is often a constant exchange between the robot and its environment. The robot needs to feel its environment by using sensors and it needs to modify its environment by moving, by speaking or by executing other tasks. The first category of tools used is the library that controls the robot. This library allowed us to communicate with the environment by controlling the motors and the speaker, for instance.

We also want our robot to learn how to follow a corridor or how to estimate a distance by judging the changes in a sequence of images. In order to perform the learning task, we used neural networks. Two kinds of neural networks were used. A simple Pattern Associator was used to learn linearly separable problems like a corridor following. In order to learn more complicated tasks, like distance estimation, we used a multi-layer Perceptron to perform the learning because it has a more general mapping property.

We also used vision tools to process images which come from the CCD camera. We used a Difference of Gaussian edge detector to compute an edge-detected image. This image can then be processed by other tools to extract features from it. We could then use histograms to detect vertical and horizontal lines. This gives us features for the corridor following task when the camera looks in a fixed direction. When the camera moves with the robot, we have to preprocess the images with a Hough transform and we can then use histograms in order to bring good features to the neural networks.

These tools were combined in order to perform two tasks. A corridor follower has been successfully built by learning features of images extracted by the histogram of an edge-detected image. We also built a robot that is able to estimate its own motion using its vision system. The program learned how to map the values that were returned by the odometry system with the visual sequences of pictures taken by the robot.

5.2 Conclusions

We have reached our goal which was to build a robot that can follow a corridor and estimate the travelled distance. The corridor follower done is very robust because the neural network used was able to use different features to do the same task. This is a general possible behaviour when we use a learning algorithm. If we build an algorithm based on landmarks, the learning algorithm can be more robust because it uses all the meaning extracted from the features and in particular it can use more landmarks that the programmer did. This behaviour adds backup to the original algorithm.

In the other hand, the estimation of the odometry using visual input that was built shows that it is possible to estimate travelled distances. Indeed, the results can seem poor because we used the internal odometry of the robot to teach it how to perceive distances. This internal odometry already makes near ten percents of relative error. We should have better results with a more accurate measure of the travelled distances during the learning.

A strong advantage of using visual input to estimate distances is that the absolute error increases slowly with the travelled distance compared to the absolute error of the odometry. Therefore, after a while the absolute error given by our estimation is less than the absolute error given by the odometry system. So we can use the odometry system to estimate the travelled distance at the beginning of a test and after a while we swap to the visual estimation of the distance. This gives us a better measure of the travelled distance.

5.3 Future work

In practice, a lot of parameters can reduce the accuracy of the results such as: the numbers of pixels in the images, the size of the stripes used for the histogram, the architecture and the size of the neural network, the orientation of the camera and also the accuracy of the odometry. The study of the exact effect of all those parameters can be an useful future work. Especially the change in the environment could be useful to study. These studies could lead to a more general distance estimator that could perform its task everywhere.

The use of a more accurate measure of the distance during the learning stage could also be investigate. We can use other devices or simply use techniques such as the one described in [9] or such as the one described in [6] designed to improve the accuracy of the internal odometry.

In order to build a robot that can follow corridors using a map provided by the user, the robot must not only follow corridors and estimate distances, but it must also be able to estimate its steering angle. A possible future work is to estimate angles when the robot turns. A first step could be to estimate angles when the robot turns without moving. The further step needed to achieve a robot that is able to navigate is to combine the angle estimation and the distance estimation. This will allow a user to build paths for the robot such as: move 10 m forward, then turn 90° on the left and move 2 m forward.

As we can see, visual odometry is an open field and there are a lot of work to do to improve the existing results and to provide an usable system.

Bibliography

[1] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming, Theory and Algorithms.* John Wiley & Sons, Inc., second edition, 1993.

[2] S. S. Beauchemin and J. L. Barron. The computation of optical flow. *ACM Computing Surveys*, 27(3):433–467, 1995.

[3] M. Bertozzi, A. Broggi, and A. Fascioli. Vision-based intelligent vehicles: State of the art and perspectives. *Robotics and Autonomous Systems*, 32:1–16, 2000.

[4] Christopher M. Bishop. *Neural networks for pattern recognition.* Oxford University Press, 1995.

[5] S. Cameron, S. Grossberg, and F. H. Guenther. A self-organizing neural network architecture for navigation using optic flow. Technical report, Boston University Center for Adaptive Systems, Department of Cognitive and Neural Systems, Boston, MA, December 1995.

[6] Stella Cicirelli Distante. Self-location of a mobile robot with uncertainty by cooperation of an heading sensor and a CCD TV camera.

[7] A. P. Duchon, W. H. Warren, and L. P. Kaelbling. Ecological robotics: Controlling behavior with optical flow. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 164–169, 1994.

[8] T. Duckett and U. Nehmzow. Quantitative analysis of mobile robot localisation systems. Technical Report UMCS-97-9-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, UK, September 1997.

[9] C. Facchinetti, F. Tièche, and H. Hügli. Three vision-based behaviors for self-positioning a mobile robot. In *Proceedings of Intelligent Autonomous Systems*, March 1995.

[10] O. Faugeras. *Three-Dimensional Computer Vision, A Geometric Viewpoint.* The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

[11] Flannery, Press, Teukolsky, and Vetterling. *Numerical recipes in C : The art of scientific computing.* Cambridge University Press, 1988.

[12] M. O. Franz and H. A. Mallot. Biomimetic robot navigation. *Robotics and Autonomous Systems*, 30:133–153, 2000.

[13] S. A. Huber, M. O. Franz, and H. H. Bülthoff. On robots and flies: Modeling the visual orientation behavior of flies. *Robotics and Autonomous Systems*, 29:227–242, 1999.

[14] Nomadic Techonlogies Inc. `http://www.robots.com`, 1999.

[15] D. Kortenkamp, M. Huber, F. Koss, W. Belding, J. Lee, A. Wu, C. Bidlack, and S. Rodgers. Mobile robot exploration and navigation of indoor spaces using sonar and vision. In *AIAA/NASA Conference in Intelligent Robots in Field, Factory, Service, and Space*, 1994.

[16] P. C. Martin. *Real Time Image Processing For Mobile Robot Control.* MSc Thesis, University Of Manchester, 1995.

[17] J. Matas, C. Galambos, and J. Kittler. Progressive probabilistic Hough transform. In *British Machine Vision Conference*, pages 256–265, 1998.

[18] M. Minoux. *Mathematical Programming, Theory and Algorithms.* Series in discrete mathematics and optimization. Wiley-Interscience Publication, 1986.

[19] Tom M. Mitchell. *Machine Learning.* Computer Science. McGraw-Hill, 1997.

[20] Ulrich Nehmzow. *Mobile Robotics: A Practical Introduction.* Applied Computing. Springer, 2000.

[21] Nomadic Technologies Inc. *Nomad 200 User's Manual*, 1995. Software version 2.6.

[22] Nomadic Technologies Inc. *Scout Beta 1.1 User's Manual*, 1998. Software version 2.6.

[23] G. M. Quénot. The "orthogonal algorithm" for optical flow detection using dynamic programming. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, pages 249–252, NY, USA, March 1992. IEEE New York.

[24] Nicholas Roy and Sebastian Thrun. Online self-calibration for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1999.

[25] G. Sandini, J. Santos-Victor, F. Curotto, and S. Garibaldi. Robotic bees. Technical report, University of Genova, Laboratory for Integrated Advances Robotics, October 1992.

[26] J. Santos-Victor, G. Sandini, F. Curotto, and S. Garibaldi. Divergent stereo in autonomous navigation: From bees to robots. *International Journal of Computer Vision*, 14:159–177, March 1995.

[27] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis, and Machine Vision.* Brooks/Cole Publishing Company, second edition, 1999.

[28] M. V. Srinivasan, J. S. Chahl, K. Weber, S. Venkatesh, M. G. Nagle, and S. W. Zhang. Robot navigation inspired by principles of insect vision. *Robotics and Autonomous Systems*, 26:203–216, 1999.

[29] M. V. Srinivasan, S. W. Zhang, and M. Lehrer. Honeybee navigation: odometry with monocular input. *Animal Behaviour*, 56:1245–1259, April 1998.

[30] D. Terzopoulos and T. F. Rabie. Animat vision: Active vision in artificial animals. In *Proceedings of the Fifth International Conference on Computer Vision*, pages 801–808. IEEE Computer Society Press, June 1995.

[31] N. D. Tillett, T. Hague, and J. A. Marchant. A robotic system for plant-scale husbandry. *Journal of Agricultural Engineering Research*, 69:169–178, 1998.

[32] K. Weber, S. Venkatesh, and D. Kieronska. Insect based navigation and its application to the autonomous control of mobile robots. In *Third International Conference on Automation, Robotics and Computer Vision*, 1994.