1986 Year End Report for Road Following at Carnegie Mellon

Charles Thorpe and Takeo Kanade
Principal Investigators

CMU-RI-TR-87-11

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

May 1987

This research was supported by the Strategic Computing Initiative of the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5351, and monitored by the U.S. Army Engineer Topographic Laboratories under contract DACA76-85-C-0003.

Table of Contents

Abstract

This report describes progress in vision and navigation for outdoor mobile robots at the Carnegie Mellon Robotics Institute during 1986. This research was sponsored by DARPA as part of the Strategic Computing Initiative.

Our work during 1986 culminated in two demonstration systems. The first system drives the Terregator, a desk-sized robot with six wheels, around the network of campus sidewalks. This system, named Sidewalk II, uses a video camera to follow sidewalks and a laser rangefinder to detect and avoid stairs. Sidewalk II makes extensive use of map data, for visual predictions and for path planning.

The second system, Park Navigation, uses the Navlab, our new Chevrolet Van robot. The Park system concentrated on vision for following difficult roads, including curves, dirt and leaves, shadows, puddles, and both moving and fixed obstacles. We developed vision techniques for handling difficult roads, and built range finder programs for detecting and avoiding obstacles.

Both the Sidewalk II and Park experiments were built into complete systems using CODGER, a novel whiteboard developed as part of the project. CODGER provides tools for handling geometry, motion over time, multiple processes, multiple processors, and multiple languages.

This report is divided into four main sections. Section 1 is an introduction and overview, including a chronology for the project and a list of 1986 publications. Section 2 describes the Sidewalk II system; section 3 describes the Park experiments, and section 4 is about CODGER.

Section I

Introduction and Overview

1. Introduction

This report reviews progress at Carnegie Mellon from January 15, 1986 to January 14, 1987 on research sponsored by the Strategic Computing Initiative of DARPA, DoD, through ARPA Order 5351, and monitored by the US Army Engineer Topographic Laboratories under contract DACA76-85-C-0003, titled "Road Following". This report consists of an introduction and overview, and three detailed reports on specific areas of research.

2. Overview

During this contract year we have built two complete systems, Sidewalk II and Park Navigation; used two robot vehicles, Terregator and Navlab; built a single underlying software system, the CODGER "whiteboard"; and transferred technology to Martin Marietta. Each of these are explained below.

A key concept in our work is integration. We have integrated data from various sensors, such as video and range, in our sensor modules. We integrate map data with perceived objects to update the vehicle's position. The whiteboard integrates separate modules into a coherent software package. And our systems integrate software, computing hardware, and mobile chassis into robots. In short, we have integrated all the separate components necessary to produce functioning mobile robots, capable of moving through difficult, realistic, outdoor scenes.

2.1. Sidewalk II

The Sidewalk II system uses information from a map, from video and range sensors, and from the Terregator's dead reckoning, to drive around the Carnegie Mellon campus sidewalks. Sidewalk II demonstrated the Terregator in continuous motion down straight paths and through intersections, and was the first actual testbed for the CMU whiteboard and system architecture. Sidewalk II perception includes low-level data fusion, building a colored range image, to recognize and locate stairs on the campus sidewalk network. The stairs are then used both as obstacles to be avoided and as landmarks for position update. Further information on Sidewalk II can be found in section II, "Sidewalk II: Perception and Capabilities." The use of the whiteboard by Sidewalk II is described in section IV, "The CMU System for Mobile Robot Navigation."

2.2. Park Navigation

The Park system drives the Navlab robot van along a winding, narrow, asphalt path through Schenley Park adjacent to the CMU campus. The focus of Park work was real world perception, both video for road following and range for obstacle avoidance. Park perception copes with difficult circumstances, including changing lighting, limited a priori models, and irregularly shaped natural objects. When the system detects an obstacle, it drives around it if possible, or if there is no clear path on the road, stops and waits for the object to move or be moved. Park navigation uses the whiteboard for system coordination. The Park navigation system is explained in detail in Section III of this report, "Vision and Navigation for the

Carnegie Mellon Navlab."

2.3. Terregator

The Terregator is the vehicle we used for all our experiments during 1985, and continues to be used for our sidewalk experiments. It is about the size of a desk, carries power and communications gear, and provides built-in motion commands. This year we have added a platform for more room, and have replaced the microwave link with two VHF video transmitters. Details of the Terregator were reported by Whittaker [2].

2.4. Navlab

The Navlab (named from "Navigation Laboratory") is a self-contained laboratory for navigational vision system research. The Navlab was based on a commercial van chassis, and is large enough to carry power, computers and researchers on board. It has been a great asset to our work to have processing and experimenters close to the action. We no longer have problems with video communications to remote computers, and researchers can quickly see the actions of their programs, and greatly speed up the debug/reprogram/test cycle. The Navlab was built under separate DARPA funding, and has been used for our Park experiments since fall 1986. The design and construction of the Navlab are chronicled by Singh [1].

2.5. Whiteboard

Intelligent mobile robots need to reason about geometrical relationships and how they change with time. A mobile robot system is built of many cooperating processes which need to communicate and to synchronize themselves. During the last year we have developed CODGER, a whiteboard, which provides tools for handling geometry, time, synchronization, and communication. On top of the CODGER tools we have built an architecture that sets conventions for control and data flow. This system structure is the basis of both the Sidewalk II and Park Navigation systems. CODGER and the associated architecture are described in Section IV of this report, "The CMU System for Mobile Robot Navigation."

2.6. Technology Transfer

Part of our charter is to cooperate with Martin Marietta in the development of the ALV (Autonomous Land Vehicle). Accordingly we have during the last year participated in the ALV quarterly meetings, in several Critical and Preliminary Design Reviews, and in a variety of less formal contacts with Martin Marietta. We have hosted a visitor from Martin for most of a year, first as a visiting scientist and since September as a graduate student. We have influenced the design of the ALV software and hardware architecture. The current combination of Suns and specialized processors on the ALV should make it relatively easy in the future to run CMU software on the ALV.

We have also contributed several stand alone modules to the ALV. Early in the year, they received a path planner that uses a terrain database of the Martin Denver site to plan paths for the ALV. In March they received code for obstacle detection using the ERIM scanner. And in October they acquired our code for terrain analysis, again using ERIM data.

3. Chronology

January:	Adaptive color runs
February:	Color cone finding
February:	First prototype whiteboard system runs
February:	Color—ERIM registration
March:	Terregator using ERIM runs in coal mine
March:	Navlab runs under joystick control
April:	First color segmentation run using Navlab with remote computers
June:	Hosted Blackboard workshop
August:	Navlab runs for the first time with on board computing, using ERIM
August:	FIDO stereo runs on Warp
October:	ERIM terrain analysis software exported to Martin Marietta
October:	Sidewalk II navigates complete course, including 90 and 135 degree turns, with continuous motion
October:	Whiteboard runs on Navlab
October:	First Navlab run with on board vision
November:	First vision runs using texture
November:	Successful runs stopping for obstacles and restarting
November:	DARPA demo of Navlab Park system
December:	Sidewalk II drives Terregator successfully around stairs

4. Personnel

Faculty: Martial Hebert, Katsushi Ikeuchi, Takeo Kanade, Steve Shafer, Chuck Thorpe, Jon Webb, William Whittaker.

Staff: Paul Allen, Mike Blackwell, Tom Chen, Jill Crisman, Kevin Dowling, Ralph Hyre, Jim Moody, Tom Palmeri, Eddie Wyatt.

Visiting scientists: Arun Agarwal, Yoshi Goto, Take Fujimori, Kichie Matsuzaki, Taka Obatake

Graduate students: Keith Gremban, Karl Kluge, InSo Kweon, Doug Reece, Bruno Serey, Tony Stentz, Rich Wallace

5. Publications

Crisman, J.
Machine Perception.
Unix Review 4(9), 1986.

Elfes, A.
A Sonar-Based Mapping and Navigation System.
In *IEEE International Conference on Robotics and Automation.* 1986.

Goto, Y., Matsuzaki, K., Kweon, I., and Obatake, T.

CMU Sidewalk Navigation System.
In *Fall Joint Computer Conference*. ACM/IEEE, November, 1986.

Hebert, M., and Kanade, T.
Outdoor Scene Analysis Using Range Data.
In *IEEE International Conference on Robotics and Automation*. 1986.

Kanade, T., Thorpe, C., and Whittaker, W.
Autonomous Land Vehicle Project at CMU.
In *ACM Computer Conference*. February, 1986.

Kanade, T. and Thorpe, C.
CMU Strategic Computing Vision Project Report: 1984 to 1985.
Technical Report, The Robotics Institute, Carnegie Mellon University, 1985.

Krogh, B., and Thorpe, C.
Integrated Path Planning and Dynamic Steering Control for Autonomous Vehicles.
In *IEEE International Conference on Robotics and Automation*. 1986.

Matthies, L.H., and Shafer, S.A.
Error modelling in stereo navigation.
In *Fall Joint Computer Conference*. ACM/IEEE, November, 1986.

Serey, B. and Matthies, L.
Obstacle avoidance using 1-D stereo vision.
Technical Report, Carnegie Mellon Robotics Institute, 1987.

Shafer, S., Stentz, A., Thorpe, C.
An Architecture for Sensor Fusion in a Mobile Robot.
In *IEEE International Conference on Robotics and Automation*. 1986.

Singh, J. et al.
NavLab: An Autonomous Vehicle.
Technical Report, Carnegie Mellon Robotics Institute, 1986.

Thorpe, C.
Vision and Navigation for the CMU Navlab.
In *SPIE*. Society of Photo-Optical Instrumentation Engineers, October, 1986.

Wallace, R., Matsuzaki, K., Goto, Y., Crisman, J., Webb, J., and Kanade, T.
Progress in Robot Road-Following.
In *IEEE International Conference on Robotics and Automation*. 1986.

References

[1] Singh, J. et al.
NavLab: An Autonomous Vehicle.
Technical Report, Carnegie Mellon Robotics Institute, 1986.

[2] Whittaker, W.
Terregator - Terrestrial Navigator.
Technical Report, Carnegie-Mellon Robotics Institute, 1984.

Section II

Sidewalk II : Perception and System Capabilities

Y. Goto , T. Obatake

I. Kweon, K. Matsuzaki

This section describes the perception and system capabilities of the Sidewalk Navigation System II. The Sidewalk II system architecture is described in section IV.

1. Perception Using Colored-Range Image

1.1. PERCEPTION Module Architecture for Sensor Fusion

The main effort in designing the PERCEPTION module is deciding how to combine several types of sensors and sensor data processing modules into one system, and how to make them work efficiently. We designed a hierarchical structure and a monitor module which manages all parts of the hierarchy (see figure 1).

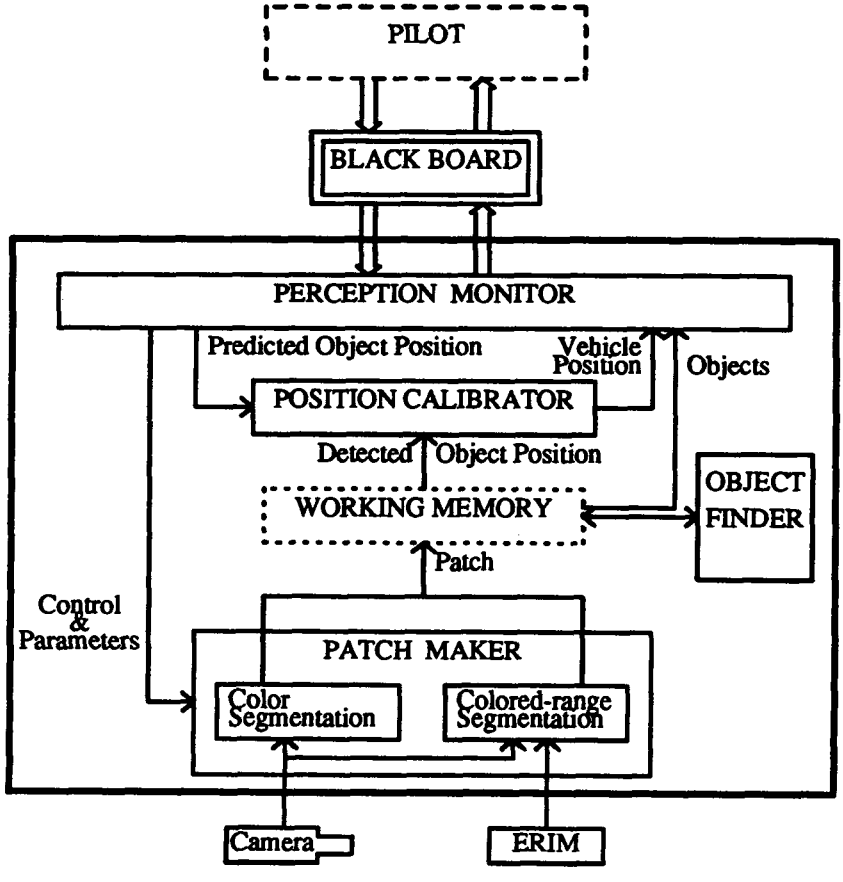


Figure 1: Structure of PERCEPTION module

1.1.1. PERCEPTION MONITOR

The PERCEPTION MONITOR has two major roles: communication with other modules (the PILOT) and control of internal submodules. The design principle of this system is to provide a common structure for different sensors and algorithms. This tends to make the module interface rather high level. For example, a desired vehicle position for image input is usually decided by an external module using sensor parameters. However, if there are several types of sensors with different view angles, the common interface for those modules will be ***where PERCEPTION should see*** instead of ***where PERCEPTION should look from***. This means the perception module itself must decide the best position from which to see the requested place. Communication with other modules means interpretation between the high level module interface commands and actual commands to internal submodules.

The control flow of the perception process is rather simple: it progresses from segmentation to position update. The PERCEPTION MONITOR activates the PATCH MAKER and the POSITION CALIBRATOR in this sequence. The functions for the interpretation of the high level commands from the other planning module (the PILOT) are described in the following paragraphs.

The PILOT requests what objects to see, but does not say which sensor should be used. The PERCEPTION MONITOR decides which sensor and segmentation module is the best for the requested objects. The current system has two sensors and segmentation modules. If all requested objects are *sidewalks* or *intersections* on a flat plane, the PERCEPTION MONITOR selects the *color segmentation* module as a PATCH MAKER. If three-dimensional objects such as stairs and slopes are included in the requested objects, the PERCEPTION MONITOR selects the *colored-range* segmentation module.

The PILOT module does not say *when* PERCEPTION should see an object because the view frame of PERCEPTION depends on the sensor used, and the PILOT does not know which sensor will be used. Instead, the PERCEPTION MONITOR uses its internal position decision algorithm.

The position decision algorithm has two steps. First, this module simulates the view frame and the vehicle's future path which is posted in the BLACKBOARD by LOCAL PATH PLANNER. When the simulated view frame covers the region which the PILOT has requested PERCEPTION to see, this vehicle position is defined as the image input position. Second, this module monitors current vehicle position by watching the moving vehicle position on the BLACKBOARD. When the moving vehicle position reaches the image input position, this module controls sensors to input an image.

1.1.2. PATCH MAKER

The PATCH MAKER, the region segmentation sub module, has a color segmentation module, a range segmentation module, and a *colored-range* segmentation module. They are described in section 1.2.

The data structure which holds Patch data segments is common to all three segmentation modules. This data includes *color type, surface type and normal, polygons for boundary shape,* and *relation to neighbor segments.*

1.1.3. POSITION CALIBRATOR

The predicted objects are described in the current coordinate system, but the vehicle coordinate system is used to describe the detected objects. The POSITION CALIBRATOR then computes the vehicle position in the current coordinate system, by applying the transformation matrix between the two

coordinate systems. The problem for this computation is that the predicted object shape and the detected object shape are not the same because of imperfections in the MAP and in perception. Therefore, the POSITION CALIBRATOR must find the most appropriate match for these two shapes.

To get the best matching point, the POSITION CALIBRATOR calculates the distance between the predicted lines and the detected lines of object polygons, and finds the position which minimizes the distance. Sometimes a scene is composed of only parallel lines (for example *sidewalk*), which are insufficient to decide a matching point. In this case, the POSITION CALIBRATOR derives a line equation on which the vehicle is located instead of a point for vehicle position.

1.2. Colored-Range Image Analysis

It is very difficult to recognize complex objects in outdoor scenes using only one kind of sensor, but several different sensors can provide many clues about the environment. For example, use of both range data and color images provides a very powerful vision system for outdoor scene analysis: range data provide information about the geometry of the scene, and color images provide information on the physical properties of objects. In order to use these different types of sensor data, we must integrate them using sensor fusion techniques. The registration between range data and color images can be a first step of sensor fusion. We call the image which has both color information and depth values a *colored-range* image. Next we describe the registration algorithm for color and range image, the segmentation procedure for range data, color segmentation algorithm, and how to use a *colored-range* image.

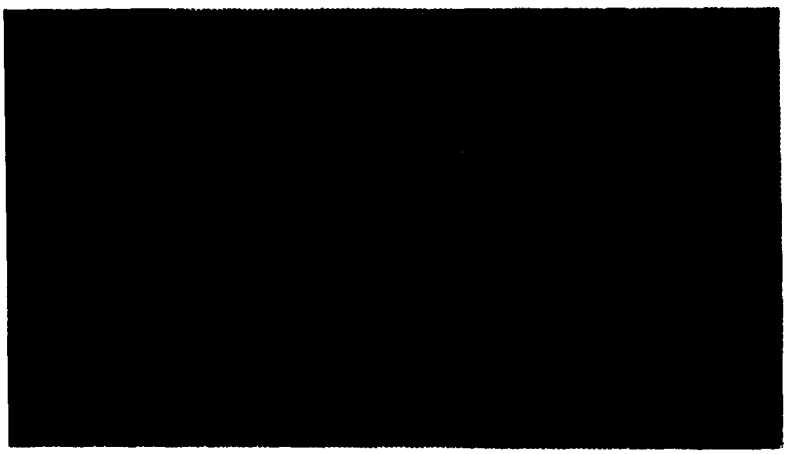
1.2.1. Registration

Colored-range images are created by registering color data onto range data. In order to register them, we need to know the camera parameters: the position and the orientation of the color camera relative to the range scanner. We developed a method to obtain these camera parameters, which consists of two steps: the initial value estimation and the optimum value estimation. The parameters calculated by the first step can be used for some simple objects. However, they are not accurate enough for our test site. The second step can give more accurate camera parameters by an iterative numerical method using the result of the first step as starting values. The details of these methods are described in the Appendix to this section.

1.2.2. Range Image Segmentation

We have two main processing modes in range segmentation: rough region segmentation and the extraction of vertical surfaces. Rough segmentation uses three basic attributes: jump edges, surface normals, and surface curvatures. Flat, horizontal surfaces can be extracted by using the surface normals, and large obstacles will be detected with surface normals pointed in other directions. This process, however, cannot provide a detailed description of a scene with small objects. In order to obtain a detailed description of a scene we need to use special purpose processing. For a scene containing stairs, we extract vertical surfaces using the fact that pixels along the column in the range image will have constant depth value. We produce the final range segmentation by combining the regions from these processes.

(a)Range image



(b)Original color image (blue intensity)

(c)*Colored-range* image

Figure 2: Color and range image registration

1.2.3. Color Segmentation

Our test site includes five types of objects to be distinguished: sidewalks, intersections, a slope, stairs, and grass. The surface color of the first four objects are almost the same shade of gray, and the grass is green. Therefore, we tried to segment the image into gray regions and non-gray regions.

The difficulties in color segmentation are caused by shadows of trees, of buildings, and of the vehicle itself, and by changing color values depending on weather conditions. In order to obtain reliable segmentation results, the program creates a 4B–G image (B: brightness of blue, G: brightness of green), which is segmented by thresholding. Subtracting green from blue reduces the effect of shadows. There are two methods for finding the threshold: When the histogram of the 4B–G image has one clear valley, the program sets the intensity value at the valley as the threshold. When it does not have a clear valley, the program selects a threshold which is close to the previous threshold value, and at which the histogram has a local minimum value.

Our color segmentation method can separate the sidewalks, the intersections, the stairs, the slope, and the grass under different weather conditions: cloudy days, rainy days, and sunny days. It also works well even if scenes include pretty sharp shadows.

1.2.4. Segmentation of Colored Range Image

The segmentation of *colored range images* is executed using both color segmentation and range image segmentation. Color segmentation assigns a color label to each pixel. Range image segmentation assigns a surface label to each pixel. Therefore, each pixel in a colored range image has both a color label and a surface label. Our method creates segments so that in each segment all pixels have the same color label and the same surface label.

1.2.5. Result of Real Scene Analysis

One good example to show the effectiveness of the *colored-range* segmentation module is a slope and stairs scene on the CMU campus sidewalk. The slope and the stairs are made of concrete and have the same gray color. The slope and roadside grass are almost on the same plane. Therefore, segmentation using only color can not separate the slope and the stairs, and segmentation using only range can not separate the slope and the road side grass. Overlap segmentation using a *colored-range* image can extract the concrete slope which is the only navigable region in this scene. Figure 3 shows the results of color segmentation, range segmentation, and final overlap segmentation.

2. System Capabilities

Using perception as described above and the system architecture described in section IV, the Sidewalk Navigation System can drive the vehicle, Terregator, on the CMU campus. It has capability

- to execute a prespecified user mission over a mapped network of sidewalks, including turning at the intersections and driving up the bicycle slope,
- to recognize landmarks, stairs and intersections under different weather conditions including sunny days, rainy days, and even if scenes include fairly sharp shadows, and
- to drive continuously at 100 mm/sec, slowing down in turning to keep turns stable.

Figure 4 illustrates the vehicle trajectories in the real runs. The vehicle navigated along the square and diagonal test course (a), and drove up the bicycle slope, avoiding the stairs (b).

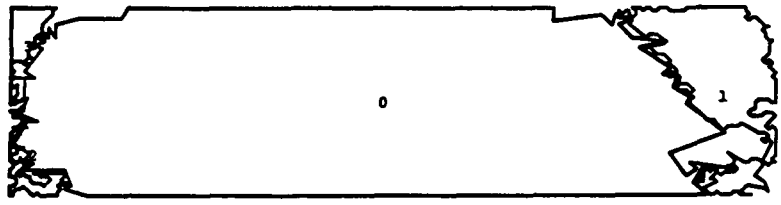
I. APPENDIX: A method for calibrating a color camera and a range scanner.

In this appendix, we describe the details of the calibration method for a color camera and a laser range scanner. The calibration consists of two steps: the initial value estimation and the optimum value estimation. We used a conventional lens calibration method to obtain a nonlinear transformation (a third order polynomial was adopted in our experiment) between the real image plane and the ideal image plane. The focal length of the camera was assumed to be unity in this experiment. Then, if we transform real image points to ideal ones using the result of the lens calibration, we can use the linear perspective projection model.

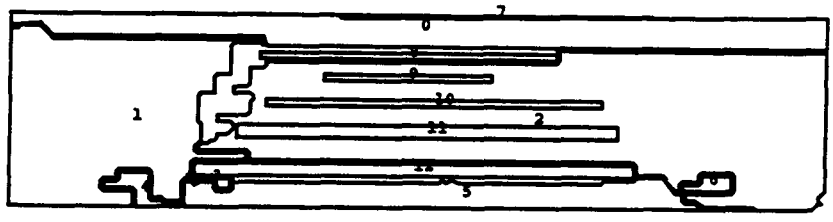
I.1. Initial estimation by a least-squares criterion

In the initial value estimation step, the measured tilt angle of the color camera is used to simplify the problem. Thus the position of the camera relative to the range scanner and its focal length are the only unknown parameters. The unknown parameters are computed by solving a least-squares criterion.

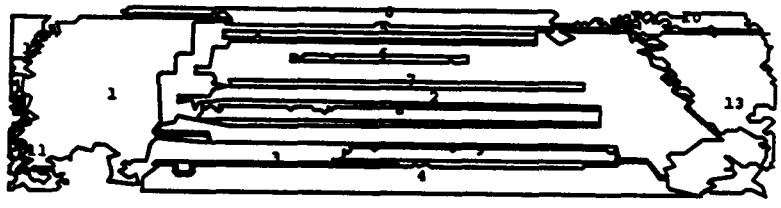
A pair of range/color images is first measured, then a set of of points $P_i^f=(x_i, y_i, z_i)$ is selected in the range image along with the corresponding set of pixels (r_i, c_i). From the homogeneous transformation, the



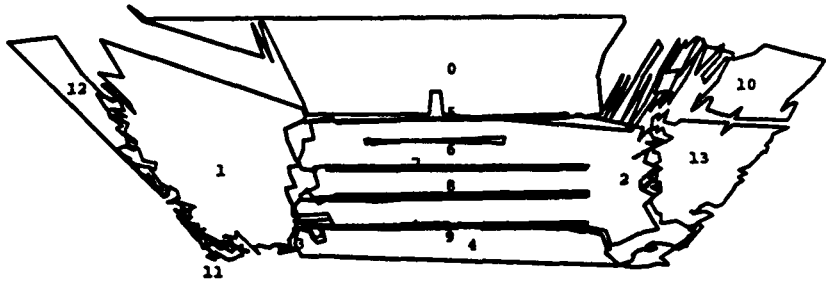
(a) Color segmentation



(b) Range segmentation



(c) *Colored-range segmentation*



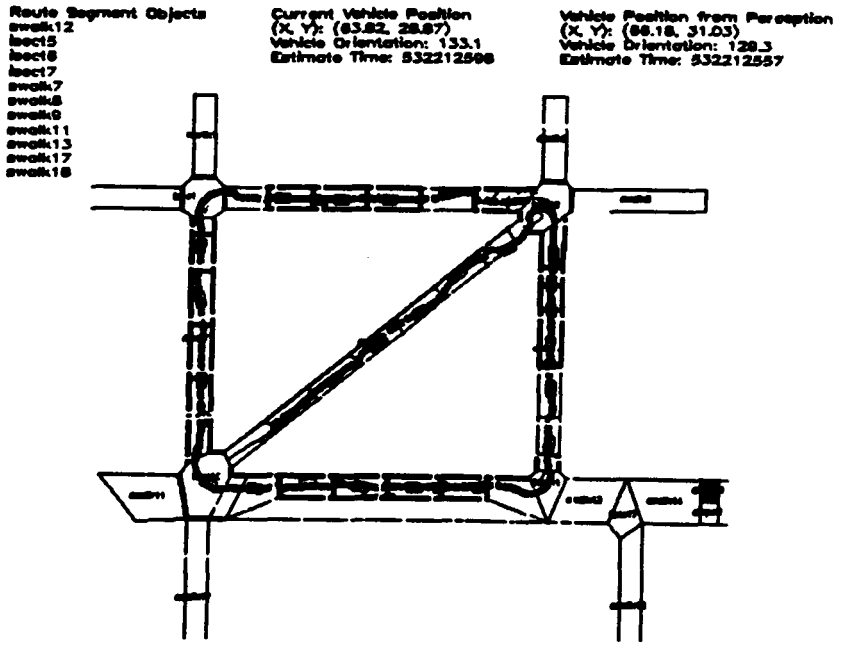
(d) *Colored-range segmentation on x-y plane*

Figure 3: Colored range segmentation

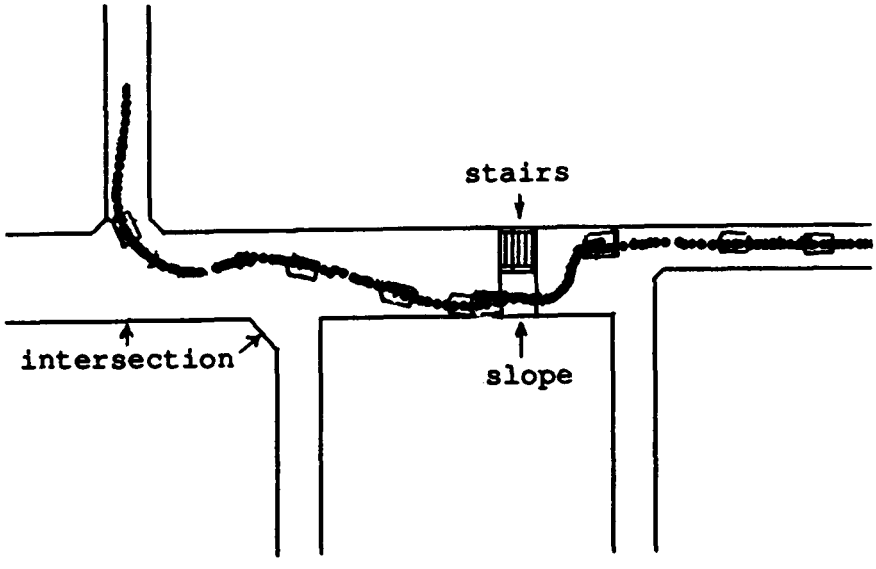
following relationship between camera-centered points and range scanner-centered points can be obtained.

$$P_i^c = R\,P_i^e - P \tag{1}$$

where, P_i^c and P_i^e are a 3D scene point in a camera-centered coordinate and a range scanner-centered coordinate respectively,



(a) Along the square and diagonal test course



(b) Around the bicycle slope and the stairs

Figure 4: The Vehicle trajectories

P is the position vector of the camera relative to the range scanner, and R is the 3x3 rotation matrix between two sensors.

Perspective transformation provides the following equations.

$$x_i^c \, r_i = f \, z_i^c \tag{2}$$

$$x_i^c \, c_i = f \, y_i^c \tag{3}$$

where, $P_i^c = (x_i^c, y_i^c, z_i^c)$ and f is the focal length of the camera.

By substituting Eq. (1) into Eq. (2) and Eq. (3) and rearranging it, we can obtain the following equations.

$$R_1 P_i^e r_i - P_x r_i - f R_3 P_i^e + P_z' = 0$$
$$R_1 P_i^e c_i - P_x c_i - f R_2 P_i^e + P_y' = 0 \quad (4)$$

where, R_1, R_2, and R_3 is each row vector of rotation matrix R and $P_y' = f P_y$, $P_z' = f P_z$

Using Eq. (4), the criterion can be written in the following form,

$$C = \sum_{i=1}^{n} \{(A_i - P_x B_i - f C_i + P_z')^2 + (D_i - P_x E_i - f F_i + P_y')^2\}$$

where, $A_i = R_1 P_i^e r_i$, $B_i = r_i$, $C_i = R_3 P_i^e$, $D_i = R_1 P_i^e c_i$, $E_i = c_i$, and $F_i = R_2 P_i^e$

To make a simpler form, we use a matrix representation.

$$C = \|U - A V\|^2 + \|W - B V\|^2 \quad (5)$$

where, $V = [P_x\ P_y'\ P_z'\ f]^t$, $A = [B_i\ 0\ -1\ C_i]$, $U = [A_1 \cdots A_n]^t$, and $W = [D_1 \cdots D_n]^t$

The camera parameters, which minimize the criterion, can be determined in the following form by taking the partial derivative of Eq. (5) with respect to the vector V and setting it to zero.

$$V = (A^t A + B^t B)^{-1} (A^t U + B^t W) \quad (6)$$

The problem of finding camera parameters is now just a matrix computation and the matrix form is given in Eq. (5).

Because the focal length is fixed as unity in our projection model, the partial derivative with respect to f is equal to zero. This causes the following changes in A, U, B, and W of Eq. (5).

$V = [P_x\ P_y'\ P_z']^t$
$A = [B_i\ 0\ -1]$
$B = [E_i\ 0\ -1]$
$U = [A_1 - C_1 \cdots A_n - C_n]^t$
$W = [D_1 - F_1 \cdots D_n - F_n]^t$

I.2. Optimum camera parameters by Newton-Gauss method

Once the initial camera parameters are computed by the first step, the optimum camera parameters including positions and orientations of the camera can be numerically computed by using the Newton-Gauss method.

The vector V has the unknown parameters as its elements. The unknown parameters are the position vector ($P_x\ P_y\ P_z$) of the camera relative to the range scanner, the pan (α), the tilt (β), and the rotation (γ) angle of the camera and the focal length f. With the measured color image points ($r_i\ c_i$), the corresponding 3-D position vectors ($x_i\ y_i\ z_i$) and the initial parameters computed by the first step, we can easily find a function $F_i(V)$ which represents the relationship between the given 3-D scene points and the corresponding color image points. The error between the measured color image points and the ideal image points can be expressed by the following equation.

$$C = \sum_{i=1}^{n} \|I_i - F_i(V)\|^2$$

$$= \sum_{i=1}^{n} \| I_i - F_i(V_0) - J_i \Delta V \|^2 \tag{7}$$

where,
$I_i = [\, r_i, c_i \,]$,
$V = [\, P_x' \; P_y' \; P_z \; f \; \alpha \; \beta \; \gamma \,]$,

$F_i(V) = [\, (f x_i^c - P_x') / (z_i^c - P_z), (f y_i^c - P_y') / (z_i^c - P_z) \,]$ and
J_i is the Jacobian matrix of a function F_i

The condition for minimum error value is that the partial derivative of Eq. (7) with respect to camera parameters should be equal to zero. From this condition, we obtain the following equation.

$$J_i^t J_i \, \Delta(V) + J_i^t \, \Delta(C) = 0 \tag{8}$$

where, J_i^t is Jacobian of function F_i,
$\Delta(V)$ is correction for camera parameters,
and $\Delta(C) = -(I_i - F_i(V_o))$

Finally the equation for the correction of camera parameters can be obtained as

$$\Delta(V) = \sum_{i=1}^{n} -(J_i^t J_i)^{-1} J_i^t \, \Delta(C) \tag{9}$$

Using Eq. (9) the procedure is iterated until there is no change in the correction values of the camera parameters.

Section III

Vision and Navigation for the Carnegie Mellon Navlab

Charles Thorpe
Martial Hebert
Takeo Kanade
Steven Shafer
and the members of
the Strategic Computing Vision Lab

1. Introduction

Robotics is where Artificial Intelligence meets the real world. AI deals with symbols, rules, and abstractions, reasoning about concepts and relationships. The real world, in contrast, is tangible, full of exceptions to the rules, and often stubbornly difficult to reduce to logical expressions. Robots must span that gap. They live in the real world, and must sense, move, and manipulate real objects. Yet to be intelligent, they must also reason symbolically. The gap is especially pronounced in the case of outdoor mobile robots. The outdoors is constantly changing, due to wind in trees, changing sun positions, even due to a robot's own tracks from previous runs. And mobility means that a robot is always encountering new and unexpected events. So static models or preloaded maps are inadequate to represent the robot's world.

The tools a robot uses to bridge the chasm between the external world and its internal representation include sensors, image understanding to interpret sensed data, geometrical reasoning, and a concept of time and of the vehicle's motion over time. We are studying those issues by building a mobile robot, the Carnegie Mellon Navlab, and giving it methods of understanding the world. The Navlab has perception routines for understanding color video images and for interpreting range data. CODGER, our *whiteboard*, proposes a new paradigm for building intelligent robot systems. The CODGER tools, developed for the Navlab and its smaller cousin the Terregator, handle much of the modeling of time and geometry, and provide for synchronization of multiple processes. Our architecture coordinates control and information flow between the high-level symbolic processes running on general purpose computers, and the lower-level control running on dedicated real-time hardware. The system built from these tools is now capable of driving the Navlab along narrow asphalt paths near campus while avoiding trees and pausing for joggers that get in its way.

This report describes the Navlab [Singh 86] and the software we have built over the past year: color vision, for finding and following roads [Thorpe 86]; 3-D perception, for obstacle avoidance [Hebert 86]; and the CODGER whiteboard [Shafer 86].

2. Navlab: Navigation Laboratory

The Navigation Laboratory, Navlab, is a self-contained laboratory for navigational vision system research (see figures 1 and 2). The motivation for building the Navlab came from our earlier experience with the Terregator, a six-wheeled vehicle teleoperated from a host computer through a radio link. The

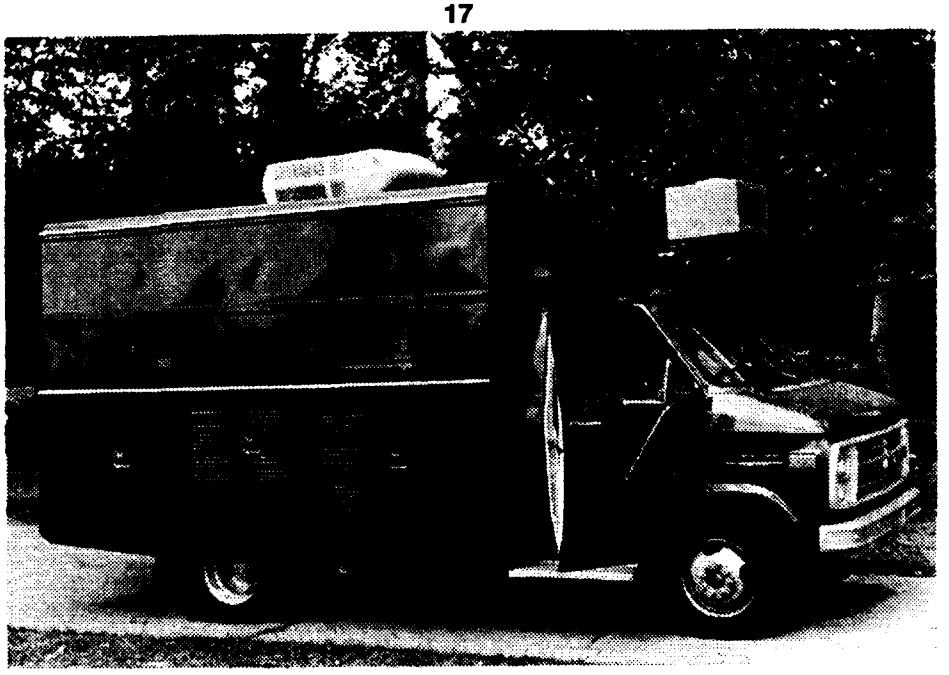


Figure 1: The Navlab

Terregator had been a reliable workhorse for small-scale experiments, such as the Campus Sidewalk navigation system [Goto 86]. However, we have outgrown its capabilities. As we began to experiment with sensor fusion, the Terregator ran out of space and power for multiple sensors. When we wanted to expand our test areas, communications to a remote computer in the lab became more difficult. And as the experiments became more sophisticated, we found it more productive for the experimenters to test or debug new programs near or in the vehicle, instead of in a remotely located laboratory. All these factors culminated in the design and construction of the Navlab [Singh 86].

Navlab is based on a commercial van chassis, with hydraulic drive and electric steering. Computers can steer and drive the van by electric and hydraulic servos, or a human driver can take control to drive to a test site or to override the computer. The Navlab has room for researchers and computers on board, and has enough power and space for all our existing and planned sensors. This gets the researchers close to the experiments, and eliminates the need for video and digital communications with remote computers.

Features of the Navlab include:

- **Onboard computers:** We have five computer racks, one for low-level controllers and power smoothing, one for video distribution, VCRs, communications and miscellaneous equipment, two racks for general-purpose processors (currently Sun workstations), and one for a Warp processor.
- **Onboard researchers:** There is always a safety driver in the driver's seat. There is room for four researchers in the back, with a terminal or workstation for each. An overhead shelf holds video monitors and additional terminals. The researchers can monitor both their programs and the vehicle's motion.
- **Onboard power:** The Navlab carries two 5500-W generators, plus power conditioning and battery backup for critical components.
- **Onboard sensors:** Above the cab is a pan mount carrying our laser scanner and a mounting rail for a color TV camera. There will eventually be a separate pan/tilt mount for stereo cameras.
- **Evolving controller:** The first computer controller for the Navlab is adequate for our current

Figure 2: Navlab interior

needs. It steers the van along circular arcs, and has commands to set speed and acceleration, and to ask for the current dead reckoned position estimate. The controller will evolve to do smoother motion control, and to interface with an inertial guidance system possibly even with GPS satellite navigation. It will also eventually watch vital signs such as computer temperature and vehicle hydraulic pressure.

3. Color Vision

The Navlab uses color vision, specifically multi-class adaptive color classification, to find and follow roads. Image points are classified into "road" or "non-road" principally on the basis of their color. Since the road is not a uniform color, color classification must have more than one road model, or class, and more than one non-road class. Because conditions change from time to time and from place to place over the test course, the color models must be adaptive. Once the image is classified, the road is identified by means of an area-based voting technique that finds the most likely location for the road in the image.

3.1. Vision Principles for the Real World

We based the development of our vision system on the following principles:

Assume variation and change. On sunny days there are shadowed areas, sunlit areas, and patches with dappled sunlight. On rainy days, there are dry patches and wet patches. Some days there are wet, dry, sunny and shadowed areas all in the same image. The road has clean spots and other places covered with leaves or with drips of our own hydraulic fluid. And as the sun goes behind a cloud or as the vehicle turns, lighting conditions change. We therefore need more than one road and non-road color model at any one time, those color models must adapt to changing conditions, and that we need to process images frequently so that the change from one image to the next will be moderate.

Use few geometric parameters. A complete description of the road's shape in an image can be complex. The road can bend gently or turn abruptly, can vary in width, and can go up- or downhill. However, the more parameters there are, the greater the chance of error in finding those parameters. Small misclassifications in an image could give rise to fairly large errors in perceived road geometry. Furthermore, if all the road parameters can vary, there are ambiguous interpretations: Does the road actually rise, or does it instead get wider as it goes? We describe the road with only two free parameters: its orientation and its distance from the vehicle. Road width is fixed, we assume a flat world, and we decree that the road is straight. While none of these assumptions is true over a long stretch of the road, they are nearly true within any one image; and the errors in road position that originate in our oversimplifications are balanced by the smaller chance of bad interpretations. If our system classifies a few pixels incorrectly as road, the worst it will do is to find a slightly incorrect road. A method that tries to fit more parameters, on the other hand, may interpret parts of the road perfectly, but could find an abrupt turn or sudden slope near any bad pixels.

Work in the image. The road can be found either by projecting the road shape into the image and searching in image coordinates, or by back projecting the image onto the ground and searching in world coordinates. The problem with the latter approach comes in projecting the image onto an evenly spaced grid in the world. The points on the world grid close to the vehicle correspond to a big area in the lower part of the image; points farther away may correspond to one or a few pixels near the top. Unless one uses a complex weighting scheme, some image pixels (those at the top that project to distant world

points) will have more weight than other (lower) points. A few noisy pixels can then have a big or a small effect, depending on where in the image they lie. On the other hand, working directly in the image makes it much easier to weight all pixels evenly. We can directly search for the road shape that has the most road pixels and the fewest non-road pixels. Moreover, projecting a road shape is much more efficient than back projecting all the image pixels.

Calibrate directly. A complete description of a camera must include its position and orientation in space, its focal length and aspect ratio, lens effects such as fisheye distortion, and nonlinearities in the optics or sensor. The general calibration problem of trying to measure each of these variables is difficult. It is much easier, and more accurate, to calibrate the whole system than to tease apart the individual parameters. The easiest method is to take a picture of a known object and build a lookup table that relates each world point to an image pixel and vice versa. Projecting road predictions into the image and back projecting detected road shapes onto the world are done by means of table lookup (or table lookup for close-by values with simple interpolations). Such a table is straightforward to build and provides good accuracy, and there are no instabilities in the calculations.

Use outside constraints. Even without a map of our test course or an expensive inertial navigation system, we know, based on the previous image and on vehicle motion, approximately where the road should be. Our *whiteboard*, described in section 5, can predict where the road should appear if the road were straight and vehicle navigation were perfect. Adding a suitable margin for curved roads and sloppy navigation still gives useful limits on where in the image to look for the road.

Test with real data. We ran our VCR nearly every time we took the vehicle out, to collect images under as many conditions as possible. We recorded sunny days, cloudy days, rainy days, leaves on trees, leaves turning color, leaves falling, early morning, noon, after dusk, even a partial solar eclipse. Strategies that worked well on one set of images did not always work on the others. We selected the toughest images, ran our best algorithms and printed the classification results, changed parameters or algorithms, reran the data set, and compared results. This gave us the best chance of being methodical and of not introducing new bugs as we went. When the image processing worked to our satisfaction, we ran simulations in the lab that included the whiteboard, range processing, path planning, and a vehicle simulator, with the vision component processing stored images and interacting with the rest of the system. When the simulations worked in the lab, we moved them to the vehicle. Only after the simulations worked on the vehicle's computers, and we were sure that all necessary software was on the van, did we go into the field for real tests. Even then not everything worked, but there were many fewer bugs than there would have been without the simulations and tests.

3.2. Road Following Algorithm

We followed these principles in building and tuning adaptive color classification for following roads. Figure 3 shows a relatively simple scene to help explain our algorithm. As shown in figure 4, the algorithm involves three stages:

1. Classify each pixel.
2. Use the results of classification to vote for the best-fit road position.
3. Collect new color statistics based on the detected road and non-road regions.

Pixel classification is done by standard pattern classification. Each class is represented by the means, variances, and covariances of red, green, and blue values, and by its a priori likelihood based on

Figure 3: Original Image

expected fraction of pixels in that class. For each pixel, calculating the class to which it most likely belongs involves finding how far the pixel's values lie from the mean of each class, where distance is measured in standard deviations of that class. Figures 5 and 6 show how each pixel is classified and how well it matches.

Once each point has been classified, we must find the most likely location of the road. We assume the road is locally flat, straight, and has parallel sides. The road geometry can then be described by two parameters as shown in figure 7:

1. The intercept, which is the image column of the road's *vanishing point*. This is where the road centerline intercepts the horizon (or more precisely the vanishing line of the locally flat plane of the road; since the camera is fixed to the vehicle this vanishing line is constant independent of the vehicle's pitch, roll, and yaw). The intercept gives the road's direction relative to the vehicle.
2. The orientation of the road in the image, which tells how far the vehicle is to the right or left of the centerline.

We set up a two-dimensional parameter space, with intercept as one dimension and orientation as the other. Each point classified as road votes for all road orientation/intercept combinations to which it could belong, while nonroad points cast negative votes, as shown in figure 9. The orientation/intercept pair that receives the most votes is the one that contains the most road points, and it is reported as the road. For the case of figure 3, the votes in orientation/intercept space look like figure 10. Figure 11 shows the detected position and orientation of the road. It is worth noting that since this method does not rely on the exact local geometry of the road, it is very robust. The road may actually curve or not have parallel edges, or the segmentation may not be completely correct. But since this method does not rely on exact geometry, the answer it produces is adequate to generate an appropriate steering command.

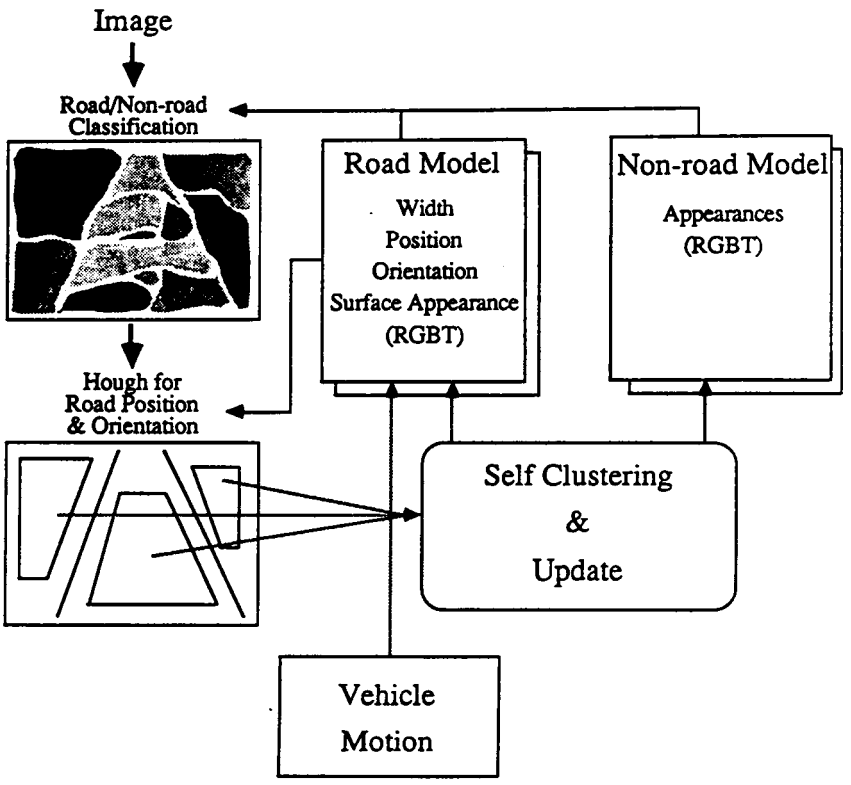


Figure 4: Color vision for road following, including color classification, Hough transform for road region detection, and updating multiple road and non-road models.

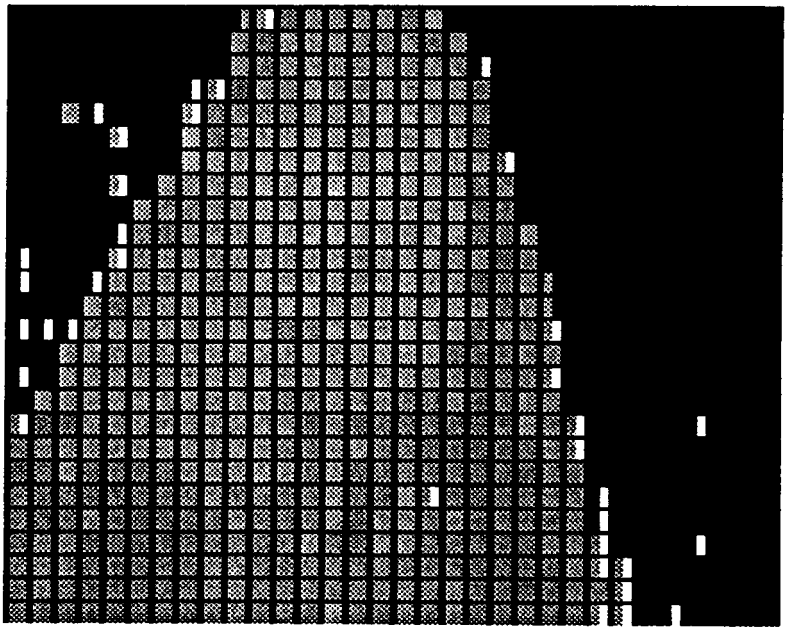


Figure 5: Segmented image. Color and texture cues are used to label points below the horizon into two road and two offroad classes

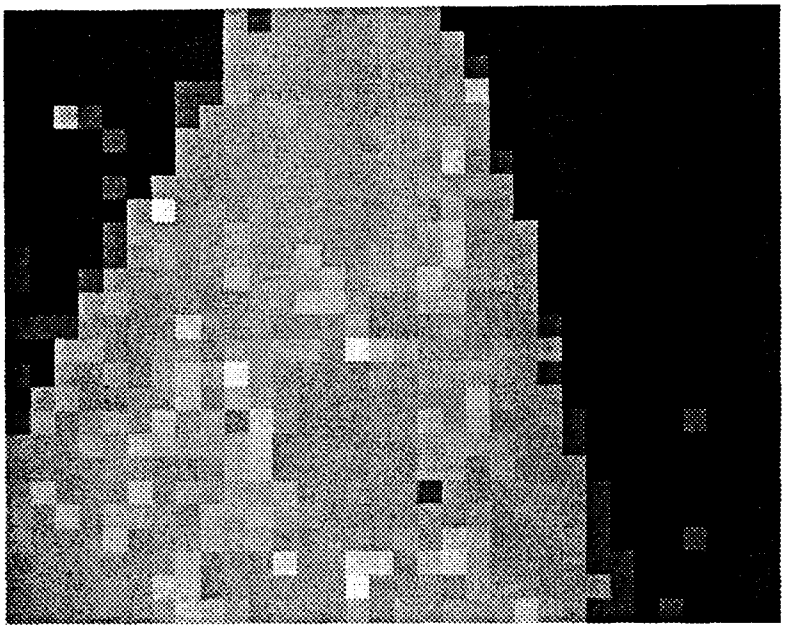


Figure 6: Road probability image. The pixels that best match typical road colors are displayed brightest.

Once the road has been found in an image, the color statistics of the road and offroad models are modified for each class by resampling the detected regions (figure 12) and updating the color models. The updated color statistics will gradually change as the vehicle moves into a different road color, as lighting conditions change, or as the colors of the surrounding grass, dirt, and trees vary. As long as the processing time per image is low enough to provide a large overlap between images, the statistics adapt as the vehicle moves. The road is picked out by hand in the first image. Thereafter, the process is automatic, using the segmentation from each image to calculate color statistics for the next.

There are several variations on this basic theme. One variation is to smooth the images first. This throws out outliers and tightens the road and non-road clusters. Another is to have more than one class for road and for non-road, for instance one for wet road and one for dry, or one for shadows and one for sun. Other variations change the voting for best road. Besides adding votes for road pixels, we subtract votes for non-road points. Votes are weighted according to how well each point matches road or non-road classes. Finally, an image contains clues other than color, such as visual texture. Roads tend to be smooth, with less high-frequency variation than grass or leaves, as shown in figure 13. We calculate a normalized texture measure, and use that in addition to color in the road classification.

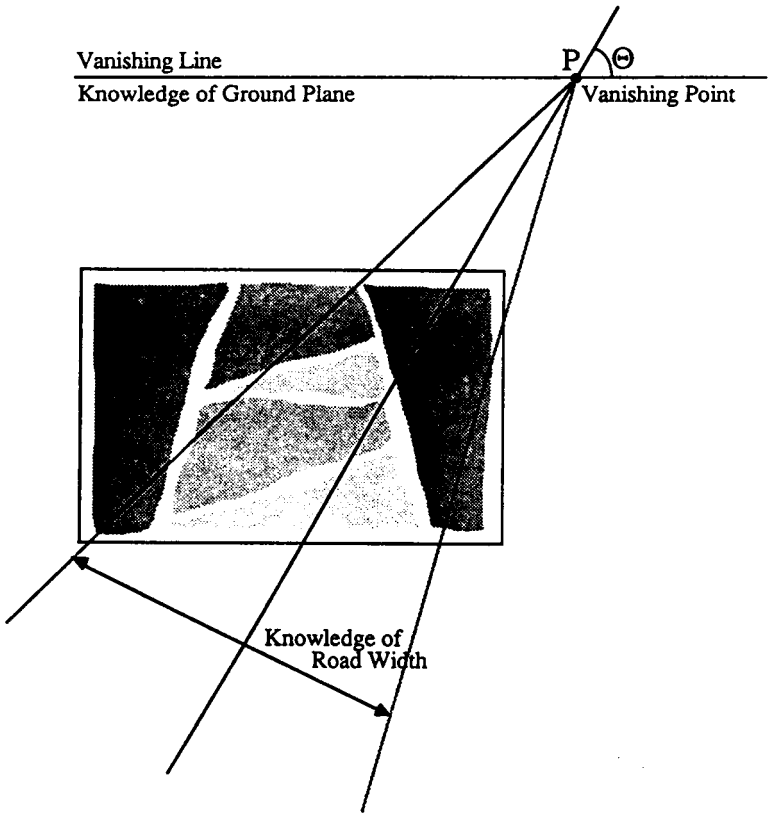
3.3. Implementation, Details, and Results

The implementation of road following runs in a loop of six steps: image reduction, color classification, texture classification, combining color and texture results, voting for road position, and color update. These steps are shown in figure 14, and are explained in detail below.

Image Reduction. We create a pyramid of reduced resolution R, G, and B images. Each smaller image is produced by simple 2 x 2 averaging of the next larger image. Other reduction methods, such as median filtering, are more expensive and produce no noticeable improvement in the system. We start

P: Road direction relative to vehicle

Θ: Vehicle position relative to road center



Find a good combination of (P,Θ)

Figure 7: Hough Transform that considers the geometry of road position and orientation. Geometry of locally flat, straight, and parallel road regions can be described by only P and θ. Point A classified as road could be a part of the road with the shown combination of (P, θ), and thus casts a positive vote for it. Point B classified as off-road, however, will cast a negative vote for that (P, θ) combination.

with 480 x 512 pixel images, and typically use the images reduced to 30 x 32 for color classification. We use less reduced versions of the images for texture classification. Image reduction is used mainly to improve speed, but as a side effect the resulting smoothing reduces the effect of scene anomalies such as cracks in the pavement.

Color Classification. Each pixel (in the 30 x 32 reduced image) is labeled as belonging to one of the road or non-road classes by standard maximum likelihood classification. We usually have two road and two non-road classes. Each class is represented by the mean R, G, and B values of its pixels, by a 3 x 3 covariance matrix, and by the fraction of pixels expected a priori to be in that class. The classification procedure calculates the probability that a pixel belongs to each of the classes, assigns the label of the most probable class, and records the maximum road and non-road probabilities for each pixel.

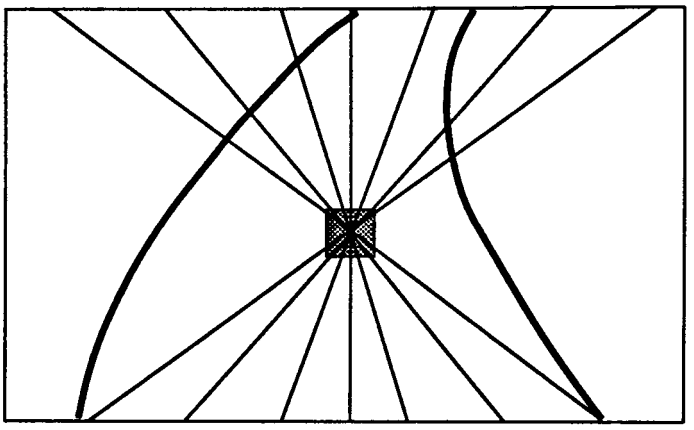


Figure 8: A road point could be a part of roads with different orientations and vanishing points.

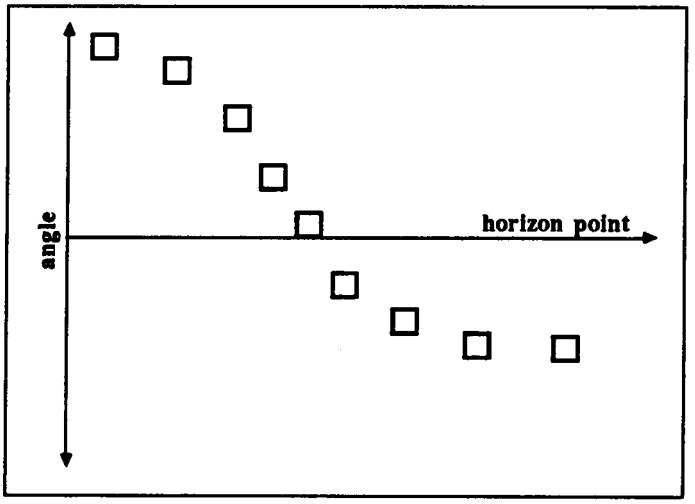


Figure 9: The point from figure 8 would vote for these orientation / intercept values.

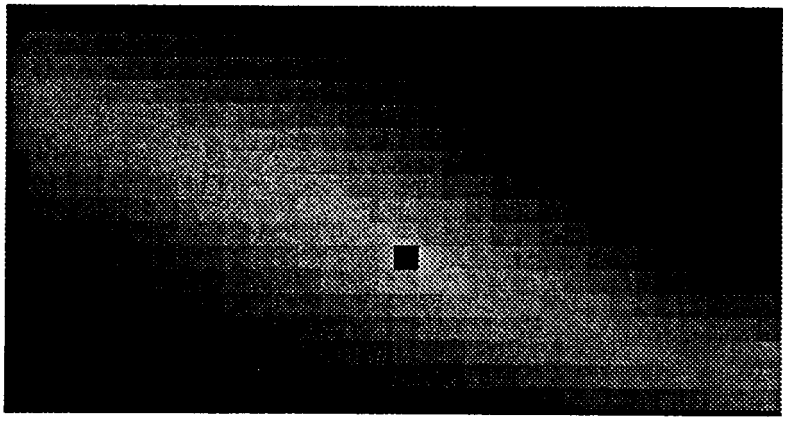


Figure 10: Votes for best road orientation and intercept, and point with most votes (dark square), for road in figure 3.

Figure 11: Detected road, from the point with the most votes shown in figure 10.

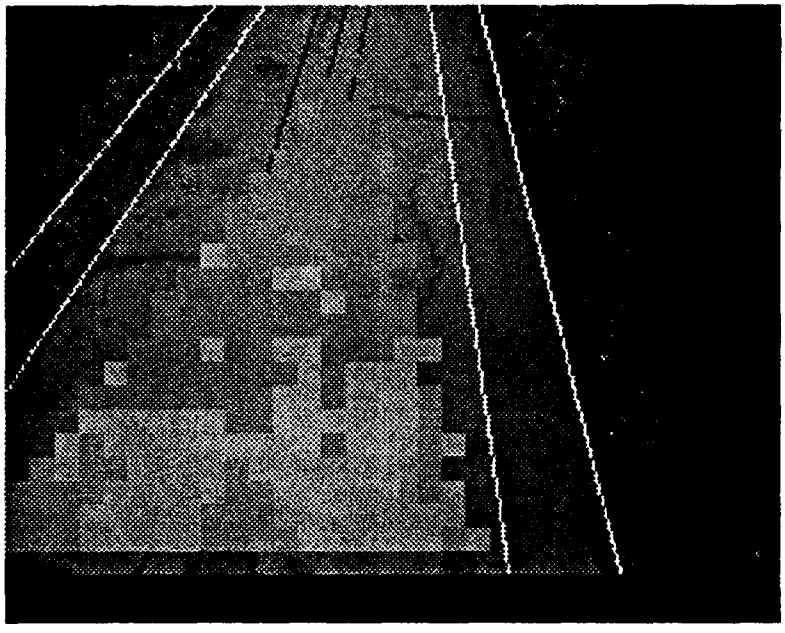


Figure 12: Updating road and nonroad model colors, leaving a safety zone around the detected road region.

Texture Calculation. This is composed of six substeps:

- Calculate texture at high resolution by running a Robert's operator over the 240 x 256 image.
- Calculate a low resolution texture by applying a Robert's operator to the 60 x 64 image.
- Normalize the texture by dividing the high resolution texture by a combination of the average pixel value for that area (to handle shadow interiors) and the low resolution texture (to remove the effect of shadow boundaries). The average pixel value is the value from the corresponding pixel in the 120 x 128 reduced image.

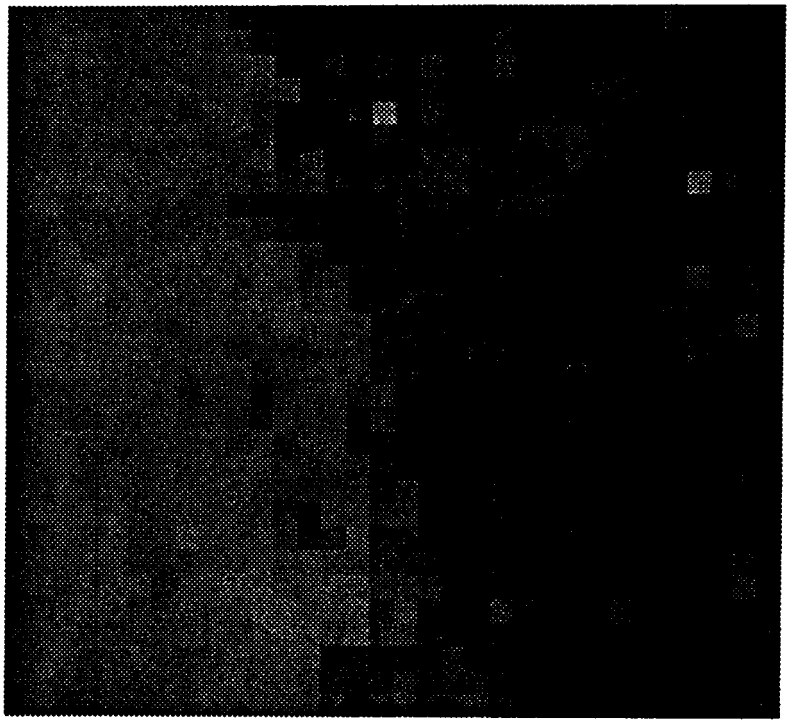


Figure 13: Zoomed picture of road-nonroad boundary. The road (at left) is much less textured than the grass (at right).

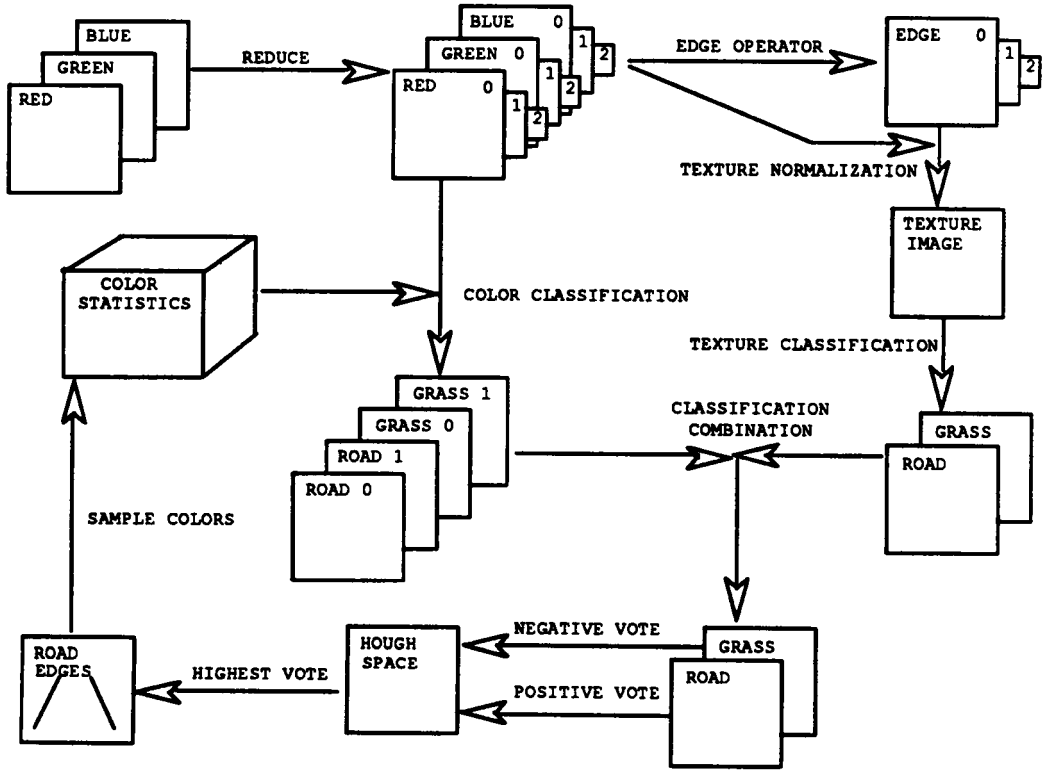


Figure 14: Processing cycle for color vision.

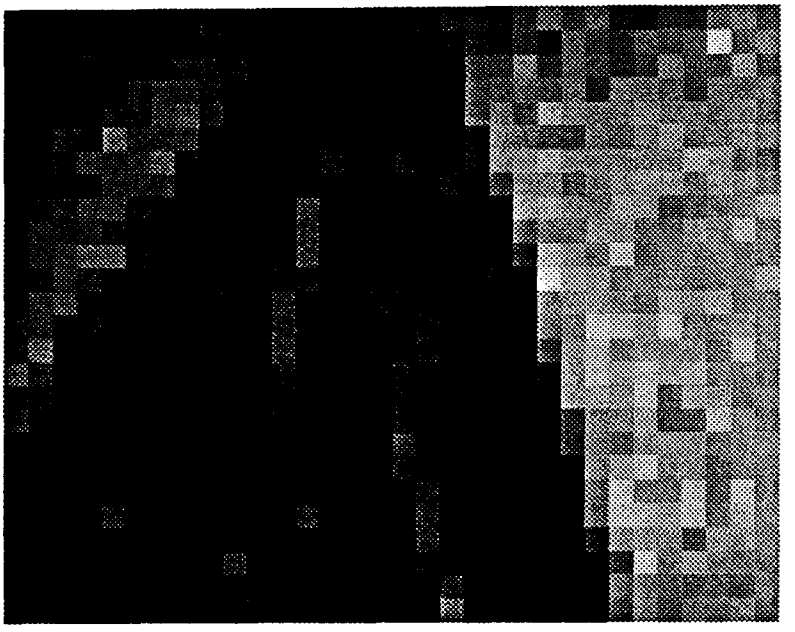


Figure 15: Low resolution texture image, showing textures from figure 3. The brighter blocks are image areas with more visual texture.

$$\text{normalized gradient} = \frac{\text{high-freq gradient}}{\alpha \times \text{low-freq gradient} + \beta \times \text{mean pixel value}}$$

Typical values for the coefficients are $\alpha = 0.2$ and $\beta = 0.8$.

- Threshold. Produce a binary image of "microedges" by thresholding the normalized gradient. A fairly low threshold, such as 1, is usually adequate.
- Count Edges. Count the number of edges in each pixel block. This gives a 30 x 32 pixel texture magnitude image. Figure 15 shows the texture image derived from figure 3. Each texture pixel has a value between 0 and 256, which is the number of pixels in the corresponding area of the full-resolution image that are microedges.
- Texture Classification. Classify each pixel in the 30 x 32 image as road or non-road on the basis of texture, and calculate a confidence for each label. We found experimentally that a fixed mean and standard deviation for road and non-road textures were better than adaptive texture parameters. Our best results were with road mean and standard deviation of 0 and 25, and non-road values of 175 and 100. Effectively, any pixel block of the image with more than 35 microedges above threshold is considered textured, and is therefore classified as nonroad.

Combination of Color and Texture Results. Color is somewhat more reliable than texture, so the color probabilities are weighted somewhat more than the probabilities calculated by texture. The result of this step is a final classification into road or non-road, and a "confidence" calculated by

Max(road confidence, non-road confidence) — Min(road confidence, non-road confidence)

Vote for Best Road Position. This step uses a 2-D parameter space similar to a Hough transform. Parameter 1 is the column of the road's vanishing point, quantized into 32 buckets because the image on which the classification and voting are based has 32 columns. Parameter 2 is the road's angle from vertical in the image, ranging from -1 to 1 radian in 0.1 radian steps. A given road point votes for all possible roads that would contain that point. The locus of possible roads whose centerlines go through

that point is an arctangent curve in the parameter space. Because the road has a finite width, the arctan curve has to be widened by the width of the road at that pixel's image row. Road width for a given row is not a constant over all possible road angles but is nearly constant enough that it doesn't justify the expense of the exact calculation. Each pixel's vote is weighted by its calculated confidence. Pixels classified as non-road cast negative votes (with their weights reduced by a factor of 0.2) while road pixels add votes. In pseudo C code, the voting for a pixel at (row, col) is

```
for (theta = -1; theta <= 1; theta+= 0.1) {
    center = col + arctan (theta);
    for (c = center - width/2; c <= center + width/2; c++) {
        parameter_space[theta][c] += confidence;
        }
    }
```

At the end of voting, one road intercept/angle pair will have the most votes. That intercept and angle describe the best road shape in the scene.

Color Update. The parameters of the road and non-road classes need to be recalculated to reflect changing colors. We divide the image into four regions plus a "safety zone": left offroad, right offroad, upper road, and lower road. We leave a 64-pixel wide "safety zone" along the road boundary, which allows for small errors in locating the road, or for limited road curvature. For each of the four regions, we calculate the means of red, green, and blue. We use the calculated parameters to form four classes, and reclassify the image using a limited classification scheme. The limited reclassification allows road pixels to be classified as either of the two road classes, but not as non-road, and allows non-road pixels to be reclassified only as one of the non-road classes. The reclassified pixels are used as masks to recalculate class statistics. The loop of classify pixels/recalculate statistics is repeated, typically 3 times, or until no pixels switch classes. The final reclassified pixels are used to calculate the means, variances, and covariances of R, G, and B for each of the classes, to be used to classify the next image. Limited reclassification is based on distance from a pixel's values to the mean values of a class, rather than the full maximum likelihood scheme used in classifying a new image. This tends to give classes based on tight clusters of pixel values, rather than lumping all pixels into classes with such wide variance that any pixel value is considered likely.

Calibration. There is no need for complete geometric calibration. The vision algorithms calculate the road's shape (road width and location of the horizon) from the first training image. We also take two calibration pictures, with a meter stick placed perpendicular to the vehicle, 8 and 12 m in front. Then during the run, given the centerline of a detected road in image coordinates, it is easy to get the x position of the road at 8 and 12 m, and then to calculate the vehicle's position on the road.

Performance. This algorithm is reliable. Running on the Navlab, with predictions of where the road should appear, our failure rate is close to 0. The occasional remaining problems come from one of three causes:

- The road is covered with leaves or snow, so one road color class and one non-road color class are indistinguishable.
- Drastic changes in illumination occur between pictures (*e.g.* the sun suddenly emerges from behind a cloud) so all the colors change dramatically from one image to the next.
- The sunlight is so bright and shadows are so dark in the same scene that we hit the hardware limits of the camera. It is possible to have pixels so bright that all color is washed out, and other pixels in the same image so dark that all color is lost in the noise.

Not every image is classified perfectly, but almost all are good enough for navigation. We sometimes find the road rotated in the image from its correct location, so we report an intercept off to one side and an angle off to the other side. But since the path planner looks ahead about the same distance as the center of the image, the steering target is still in approximately the correct location, and the vehicle stays on the road. This algorithm runs in about 10 s per image on a dedicated Sun 3/160, using 480 x 512 pixel images reduced to 30 rows by 32 columns. We currently process a new image every 4 m, which gives about three fourths of an image overlap between images. Ten seconds is fast enough to balance the rest of the system but is slow enough that clouds can come and go and lighting conditions change between images. We are porting this algorithm to the Warp, Carnegie Mellon's experimental high-speed processor. On that machine, we hope to process an image per second and to use higher resolution.

4. Perception in 3-D

Our obstacle detection starts with direct range perception using an ERIM scanning laser rangefinder. Our ERIM produces, every half second, an image containing 64 rows by 256 columns of range values; an example is shown in figure 16. The scanner measures the phase difference between an amplitude-modulated laser and its reflection from a target object, which in turn provides the distance between the target object and the scanner. The scanner produces a dense range image by using two deflecting mirrors, one for the horizontal scan lines and one for vertical motion between scans. The volume scanned is 80 degrees wide and 30 high. The range at each pixel is discretized over 256 levels from zero to 64 feet.

Figure 16: Range image of two trees on flat terrain. Gray levels encode distance; nearer points are painted darker.

Our range processing begins by smoothing the data and undoing the peculiarities of the ranging geometry. The *ambiguity intervals*, where range values wrap around from 255 to 0, are detected and unfolded. Two other undesirable effects are removed by the same algorithm. The first is the presence of mixed points at the edge of an object. The second is the meaninglessness of a measurement from a surface such as water, glass, or glossy pigments. In both cases, the resulting points are in regions limited by considerable jumps in range. We then transform the values from angle-angle-range, in scanner coordinates, to x-y-z locations. These 3-D points are the basis for all further processing.

We have two main processing modes: obstacle detection and terrain analysis. Obstacle detection starts by calculating surface normals from the x-y-z points. Flat, traversable surfaces will have vertical surface normals. Obstacles will have surface patches with normals pointed in other directions. This

analysis is relatively fast, running in about 5 s on a Sun 3/75, and is adequate for smooth terrain with discrete obstacles.

Simple obstacle maps are not sufficient for detailed analysis. For greater accuracy we do more careful terrain analysis and combine sequences of images corresponding to overlapping parts of the environment into an *extended obstacle map*. The terrain analysis algorithm first attempts to find groups of points that belong to the same surface and then uses these groups as seeds for the region growing phase. Each group is expanded into a smooth connected surface patch. The smoothness of a patch is evaluated by fitting a surface (plane or quadric). In addition, surface discontinuities are used to limit the region growing phase. The complete algorithm is:

1. Edges: Extract surface discontinuities, pixels with high jumps in x-y-z.
2. Clustering: Find clusters in the space of surface normals and identify the corresponding regions in the original image.
3. Region growing: Expand each region until the fitting error is larger than a given threshold. The expansion proceeds by iteratively adding the point of the region boundary that adds the minimum fitting error.

The clustering step is designed so that other attributes such as color or curvature can also be used to find potential regions on the object. The primitive surface used to compute the fitting error can be either a plane or a quadric surface. The decision is based on the size of the region. Figure 17 shows the resultant description of 3-D terrain and obstacles for the image of figure 16. The flat, smooth, navigable region is the meshed area, and the detected 3-D objects (the two trees) are shown as polyhedra.

Obstacle detection works at longer range than terrain analysis. When the scanner is looking at distant objects, it has a very shallow depression angle. Adjacent scanlines, separated by 0.5 degree in the range image, can strike the ground at widely different points. Because the grazing angle is shallow, little of the emitted laser energy returns to the sensor, producing noisy pixels. Noisy range values, widely spaced, make it difficult to do detailed analysis of flat terrain. A vertical obstacle, such as a tree, shows up much better in the range data. Pixels from neighboring scanlines fall more closely together, and with a more nearly perpendicular surface the returned signal is stronger and the data cleaner. It is thus much easier for obstacle detection to find obstacles than for terrain analysis to certify a patch of ground as smooth and level.

When neither video nor range information alone suffices, we must fuse data to determine mobility or recognize an object. One such case occurs in navigating the smaller Terregator vehicle around campus sidewalks. At one spot, a sidewalk goes up a flight of stairs and a bicycle path curves around. Video alone has a tough time distinguishing between the cement stairs and the cement bicycle path. Range data cannot tell the difference between the smooth rise of the grassy hill and the smooth bicycle ramp. The only way to identify the safe vehicle path is to use both kinds of data.

We start by fusing the data at the pixel level. For each range point, we find the corresponding pixel in the video image. We produce a painted range image in which each pixel is a {red, green, blue, x, y, z} 6-vector. Figure 18 shows the painted range image, rotated and projected from a different angle. We can then run our standard range segmentation and color segmentation programs, producing regions of smooth range or constant color. For the stairs in particular, we have a special-purpose step detection program that knows about vertical and horizontal planes and how they are related in typical stairs. It is

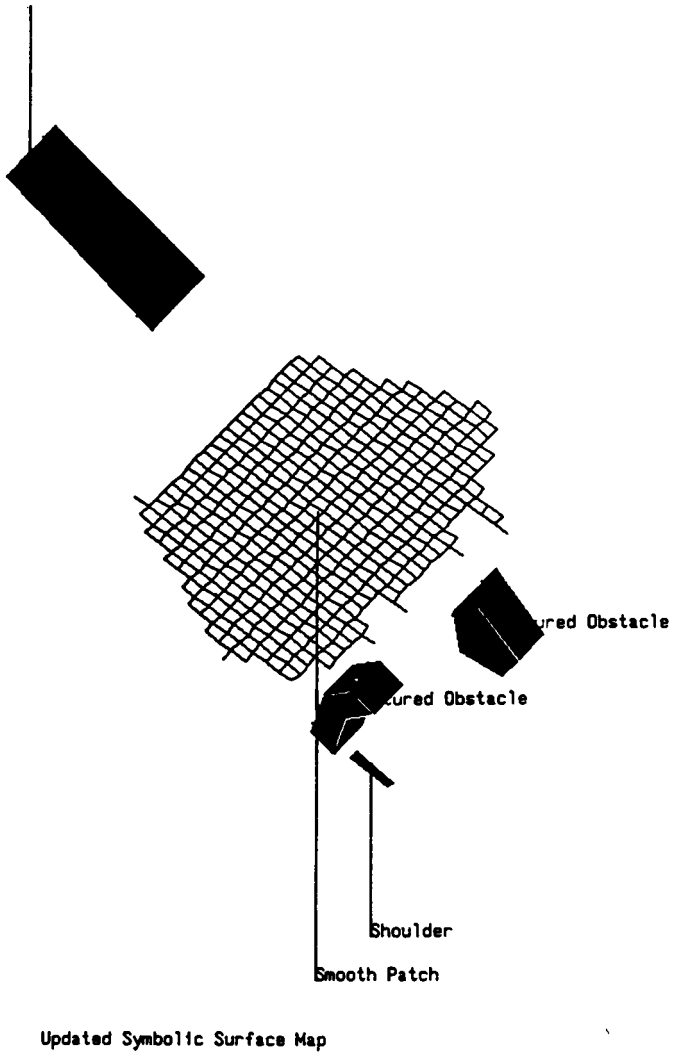


Figure 17: The resultant description of 3D terrain and obstacles from the image in figure 16. The navigable area is shown as a mesh, and the two trees are detected as "textured obstacles" and shown as black polygons

easy to combine the regions from these separate processes, since they are all in the same coordinates of the painted range image. The final result is a smooth concrete region in which it is safe to drive, and a positive identification and 3-D location of the stairs, for updating the vehicle position.

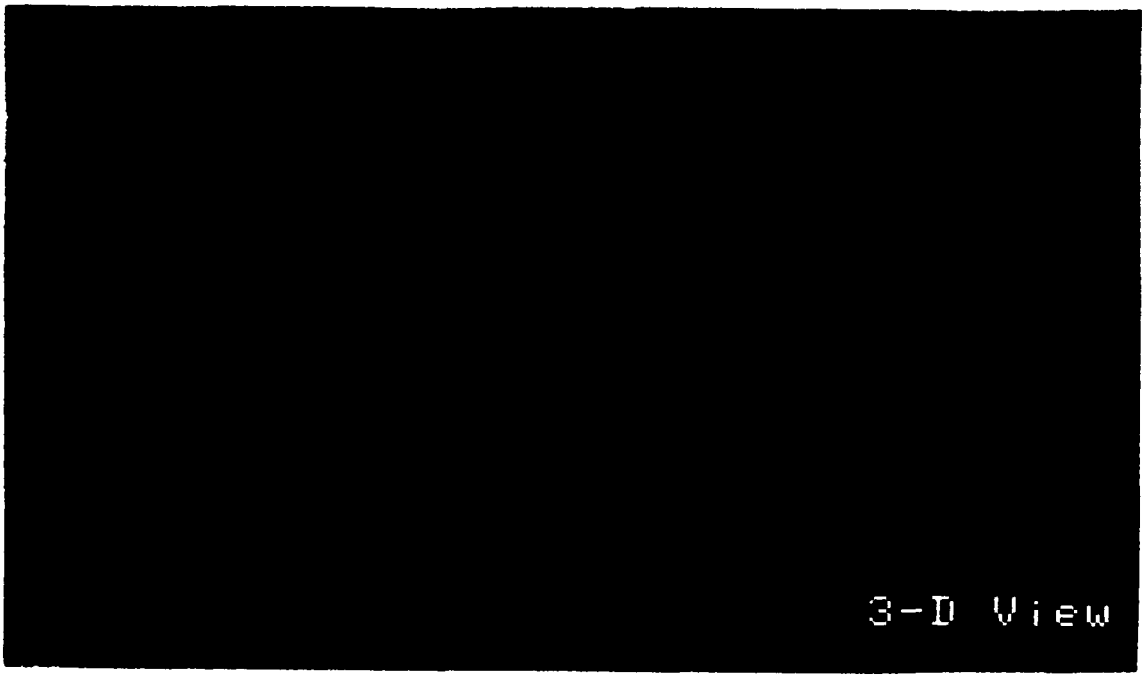


Figure 18: Painted range image of campus stairs. Each point is a {red, green, blue, x, y, z} 6-vector. This image has been rotated and projected from a different viewpoint. The color and range images are registered, so the color edges and regions line up with range edges and regions.

5. System Building

5.1. Artificial Intelligence for Real World Robots

We have developed a new paradigm for intelligent robot system building. Artificial Intelligence systems, including intelligent mobile robots, are symbol manipulators. Indeed, the very definition of intelligence, artificial or otherwise, includes symbol manipulation. But the manipulation used by most AI systems is based on inference, either by the logic of predicate calculus or by probabilities. The bulk of the work of a mobile robot, in contrast, is based on geometry and on modeling time. Inference may be a part of a mobile robot system, but geometry and time are pervasive. Consequently, intelligent mobile robots need a new kind of expert system *shell*, one that provides tools for handling 3-D locations and motion.

This fits into the context of changes in the field of AI as a whole. Early systems, such as the Logic Theorist or GPS [Cohen 82], were search engines that had no domain knowledge. They could solve problems such as the Towers of Hanoi or Missionaries and Cannibals that are essentially logic puzzles. "Expert systems" brought lots of knowledge to bear on a problem. A system such as R1 or MYCIN [Cohen 82] has thousands of rules of the form "if P then try Q" or "if X is true then Y is true with confidence 0.7". This type of knowledge allows these programs to deal with many real world problems. However, it is "shallow" knowledge in the sense that it deals with externally visible input-output behavior, with no knowledge of internal structure or mechanisms. MYCIN is like a doctor who has never taken Anatomy or Physiology, but has seen a lot of cases. Its knowledge is adequate for handling things it has already seen, but, because it does not understand the underlying mechanisms and structures of its domain, there is a limit to its competence in reasoning about new or unexpected behavior. The newest generation of expert systems is beginning to embed more "deep knowledge." For instance, the ALADIN aluminum alloy design system [Rychener 86] includes both shallow knowledge rules ("If the alloy is too heavy, try adding lithium") and deep knowledge of crystal structure and chemical interactions.

The evolution of mobile robot systems is following an analogous course. Early systems such as SRI's Shakey were based on deduction. Shakey could decide which light switch to flip and in what order to traverse a sequence of rooms; it was a success with respect to logical action, but it lacked the deep knowledge needed to move and live in a complicated environment. Its home was a series of empty rooms with flat floors and uniform walls that allowed Shakey to function with very simple perception and motion capabilities. In contrast, a robot that must move through the real outdoor world, needs a vast reservoir of deep knowledge of perception, object models, motion, path planning, terrain models, navigation, vehicle dynamics, and so forth.

The deep knowledge needed by a mobile robot must be supported by the system architecture and by the system building tools. We have developed and followed the following tenets of mobile robot system design in building our system:

Use separate modules. Much of the deep knowledge can be limited to particular specialist modules. The effects of lighting conditions and viewing angle on the appearance of an object, for instance, are important data for color vision but are not needed by path planning. So one principle of mobile robot system design is to break the system into modules and minimize the overlap of knowledge between modules.

Provide tools for geometry and time. Much of the knowledge that needs to be shared between

modules has to do with geometry, time, and motion. An object may be predicted by one module (the lookout), seen separately by two others (color vision and 3-D perception), and used by two more (path planner and position update). During the predictions, sensing, and reasoning, the vehicle will be moving, new position updates may come in, and the geometrical relationship between the vehicle and the object will be constantly changing. Moreover, there may be many different frames of reference: one for each sensor, one for the vehicle, one for the world map, and others for individual objects. Each module should be able to work in the coordinate frame that is most natural; for instance, a vision module should work in camera coordinates and should not have to worry about conversion to the vehicle reference frame. The system should provide tools that handle as many as possible of the details of keeping track of coordinate frames, motion, and changing geometry.

Provide tools for synchronization. A system that has separate modules communicating at a fairly coarse grain will be loosely coupled. Lock-step interactions are neither necessary nor appropriate. However, there are times when one module needs to wait for another to finish, or when a demon module needs to fire whenever certain data appear. The system should provide tools for several different kinds of interaction and for modules to synchronize themselves as needed.

Handle real-time vs symbolic interface. At one level, a mobile robot reasons symbolically about perceived objects and planned paths, probably on a slow time scale. At the same time, the vehicle is constantly moving, and low-level servo processes are controlling steering and motion. The top level processes need to be free to take varying amounts of time to process scenes of varying difficulty. They are often event driven, running when a particular object is seen or a particular event occurs. The servo processes, though, must run continuously and in real time (not "simulated real time" or "real time not counting garbage collection"). The system should provide for both real-time and asynchronous symbolic processes, and for communications between them.

Provide a *virtual vehicle.* As many as possible of the details of the vehicle should be hidden. At Carnegie Mellon, we have one robot (the Terregator) that has six wheels, steers by driving the wheels on one side faster than those on the other side, and carries a camera mount approximately 6 ft high. A second robot (the Navlab) is based on a commercial van, steers and drives conventionally, and mounts its camera 2 ft higher. We need to be able to use one system to drive either of the vehicles, with only minor modifications. This requires hiding the details of sensing and motion in a "virtual vehicle" interface, so a single "move" command, for instance, will use the different mechanisms of the two vehicles but will produce identical behavior.

Plan for big systems. It takes good software engineering to build a mobile robot. The system may be written in a mixture of programming languages, will probably run on multiple processors, and may use different types of processors including specialized perception machines. System tools must bridge the gaps between languages, data formats, and communications protocols.

In addition to these tenets of good design, we have identified certain approaches that are inappropriate. Many good ideas in other areas of AI present difficulties for mobile robots. Specifically, we avoid the following.

Do not throw away geometric precision. Mobile robots need all the information they can get. It is often important to know as precisely as possible where an object is located, either for planning efficient paths or for updating vehicle location. There is no need to turn a measured distance of 3.1 m into *fairly*

close. Given the relative costs and speeds of computers and vehicles, it is more efficient to spend extra computing effort (if any) to handle precise data than to plan fuzzy paths that take the vehicle unnecessarily far out of its way.

Do not concentrate on explanations. It is important to have hooks inside the vehicle's reasoning, for debugging and for learning about the system behavior. However, the prime output of the vehicle is its externally observable behavior. Producing explanations is nice, but is not the primary product as it is in expert systems for diagnosis or in intelligent assistants.

Do not build an omniscient master process. In some systems (notably early blackboards) a single master process "knows" everything. The master process may not know the internal working of each module, but it knows what each module is capable of doing. The master controls who gets to run when. The master itself becomes a major AI module and can be a system bottleneck. In contrast, the individual modules in a mobile robot system should be autonomous, and the system tools should be slaves to the modules. The module writers should decide when and how to communicate and when to execute. The system support should be as unobtrusive as possible.

We have followed these tenets in building the Navlab system. At the bottom level, we have built the CODGER "whiteboard" to provide system tools and services. On top of CODGER we have built an architecture that sets conventions for control and data flow. CODGER and our architecture are explained below.

5.2. Blackboards and Whiteboards

The program organization of the NAVLAB software is shown in figure 19. Each of the major boxes represents a separately running program. The central database, called the *Local Map*, is managed by a program known as the Local Map Builder *(LMB)*. Each module stores and retrieves information in the database through a set of subroutines called the *LMB Interface* which handle all communication and synchronization with the LMB. If a module resides on a different processor than the LMB, the LMB and LMB Interface will transparently handle the network communication. The Local Map, LMB, and LMB Interface together comprise the CODGER (COmmunications Database with GEometric Reasoning) system.

The overall system structure—a central database, a pool of knowledge-intensive modules, and a database manager that synchronizes the modules—is characteristic of a traditional blackboard system. Such a system is called "heterarchical" because the knowledge is scattered among a set of modules that have access to data at all levels of the database (i.e. low-level perceptual processing ranging up to high-level mission plans) and may post their findings on any level of the database; in general, heterarchical systems impose de facto structuring of the information flow among the modules of the system. In a traditional blackboard, there is a single flow of control managed by the database (or blackboard) manager. The modules are subroutines, each with a predetermined precondition (pattern of data) that must be satisfied before that module can be executed. The manager keeps a list of which modules are ready to execute. In its central loop it selects one module, executes it, and adds to its ready-list any new modules whose preconditions become satisfied by the currently executing module. The system is thus synchronous and the manager's function is to focus the attention of the system by selecting the "best" module from the ready-list on each cycle.

We call CODGER a *whiteboard* because although it implements a heterarchical system structure, it

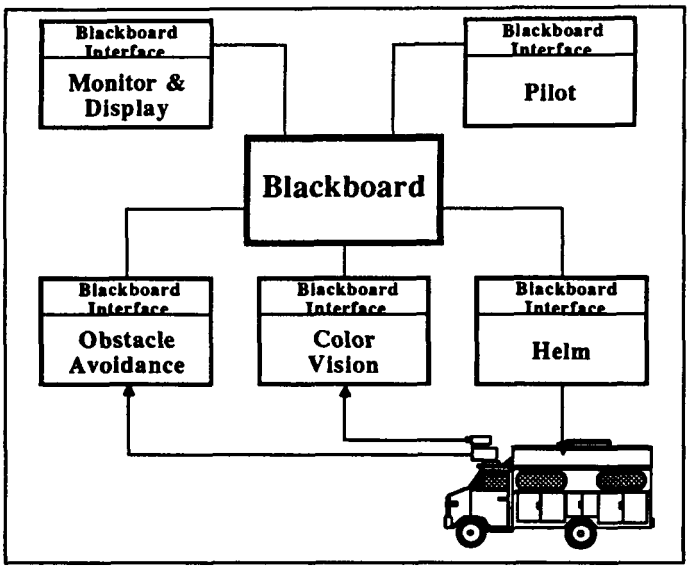


Figure 19: Navlab software architecture

differs from a blackboard in several key respects. In CODGER, each module is a separate, continuously running program; the modules communicate by storing and retrieving data in the central database. Synchronization is achieved by primitives in the data retrieval facilities that allow, for example, for a module to request data and suspend execution until the specified data appears. When some other module stores the desired data, the first module will be reactivated and the data will be sent to it. With CODGER a module programmer thus has control over the flow of execution within his module and may implement real-time loops, demons, data flows among cooperating modules, etc. CODGER also has no precompiled list of data retrieval specifications; each time a module requests data, it provides a pattern for the data desired at that time. A whiteboard is heterarchical like a blackboard, but each module runs in parallel, with the module programmer controlling the synchronization and data retrieval requests as best suited for each module. Like other recent distributed AI architectures, whiteboards are suited to execution on multiple processors.

5.3. Data Storage and Retrieval

Data in the CODGER database (Local Map) is represented in *tokens* consisting of classical *attribute-value pairs*. The types of tokens are described in a *template file* that tells the name and type of each attribute in tokens of each type. The attributes themselves may be the usual scalars (integers, floating-point values, strings, enumerated types), arrays (or sets) of these types (including arrays of arrays), or geometric locations (as described below). CODGER automatically maintains certain attributes for each token: the token type and id number, the *generation number* as the token is modified, the time at which the token was created and inserted into the database, and the time at which the sensor data was acquired that led to the creation of this token. The LMB Interface provides facilities for building and dissecting tokens and attributes within a module. Rapid execution is supported by mapping the module programmer's names for tokens and attributes onto globally used index values at system startup time.

A module can store a token by calling a subroutine to send it to the LMB. Tokens can be retrieved by constructing a pattern called a *specification* and calling a routine to request that the LMB send back tokens matching that specification. The specification is simply a Boolean expression in which the

attributes of each token may be substituted; if a token's attributes satisfy the Boolean expression, then the token is sent to the module that made the request. For example, a module may specify:

tokens with **type** *equal to "intersection" and* **traffic-control** *equal to "stop-sign"*

This would retrieve all tokens whose **type** and **traffic-control** attributes satisfy the above conditions. The specification may include computations such as mathematical expressions, finding the minimum value within an array attribute, comparisons among attributes, etc. CODGER thus implements a general database. The module programmer constructs a specification with a set of subroutines in the CODGER system.

One of the key features of CODGER is the ability to manipulate geometric information. One of the attribute types provided by CODGER is the *location*, which is a 2-D or 3-D polygon and a reference to a *coordinate frame* in which that polygon is described. Every token has a specific attribute that tells the location of that object in the Local Map, if applicable, and a specification can include geometric calculations and expressions. For example, a specification might be:

tokens with **location** *within 5 units of (45,32) [in* world *coordinates]*

or

tokens with **location** *overlapping X*

where *X* is a description of a rectangle on the ground in front of the vehicle. The geometric primitives currently provided by CODGER include calculation of centroid, area, diameter, convex hull, orientation, and minimum bounding rectangle of a location, and distance and intersection calculations between a pair of locations. We believe that this kind of geometric data retrieval capability is essential for supporting spatial reasoning in mobile robots with multiple sensors. We expect geometric specifications to be the most common type of data retrieval request used in the NAVLAB.

CODGER also provides for automatic coordinate system maintenance and transformation for these geometric operations. In the Local Map, all coordinates of location attributes are defined relative to **WORLD** or **VEHICLE** coordinates; **VEHICLE** coordinates are parameterized by time, and the LMB maintains a time-varying transformation between **WORLD** and **VEHICLE** coordinates. Whenever new information (i.e. a new **VEHICLE**-to-**WORLD** transform) becomes available, it is added to the "history" maintained in the LMB; the LMB will interpolate to provide intermediate transformations as needed. In addition to these basic coordinate systems, the LMB Interface allows a module programmer to define *local coordinates* relative to the basic coordinates or relative to some other local coordinates. Location attributes defined in a local coordinate system are automatically converted to the appropriate basic coordinate system when a token is stored in the database. CODGER provides the module programmer with a conversion routine to convert any location to any specified coordinate system.

All of the above facilities need to work together to support asynchronous sensor fusion. For example, suppose we have a vision module A and a rangefinder module B whose results are to be merged by some module C. The following sequence of actions might occur:

1. A receives an image at time 10 and posts results on the database at time 15. Although the calculations were carried out in the camera coordinate system for time 10, the results are automatically converted to the **VEHICLE** system at time 10 when the token is stored in the database.
2. Meanwhile, B receives data at time 12 and posts results at time 17 in a similar way.
3. At time 18, C receives A's and B's results. As described above, each such token will be tagged with the time at which the sensor data was gathered. C decides to use the vehicle

coordinate system at time 12 (B's time) for merging the data.

4. C requests that A's result, which was stored in **VEHICLE** time 10 coordinates, be transformed into **VEHICLE** time 12 coordinates. If necessary, the LMB will automatically interpolate coordinate transformation data to accomplish this. C can now merge A's and B's results since they are in the same coordinate system. At time 23, C stores results in the database, with an indication that they are stored in the coordinate system of time 12.

5.4. Synchronization Primitives

CODGER provides module synchronization through options specified for each data retrieval request. Every time a module sends a specification to the LMB to retrieve tokens, it also specifies options that tell how the LMB should respond with the matching tokens:

- *Immediate Request.* The module requests all tokens currently in the database that match this specification. The module will block (i.e. the "request" subroutine in the LMB Interface will not return control) until the LMB has responded. If there are no tokens that match the specification, the action taken is determined by an option in the module's request:
  - *Non-Blocking.* The LMB will answer that there are no matching tokens, and the module can then proceed. This would be used for time-critical modules such as vehicle control. Example: "Is there a stop sign?"
  - *Blocking.* The LMB will record this specification and compare it against all incoming tokens. When a new token matches the specification, it will be sent to the module and the request will be satisfied. Meanwhile, the module will remain blocked until the LMB has responded with a token. This is the type of request used for setting up synchronized sets of communicating modules: each one waits for the results from the previous module to be posted to the database. Example: "Wake me up when you see a stop sign."
- *Standing Request.* This provides a mechanism for the LMB to generate an interrupt for a running module. The module gives a specification along with the name of a subroutine. The module then continues running; the LMB will record the specification and compare it with all incoming tokens. Whenever a token matches, it will be sent to the module. The LMB Interface will intercept the token and execute the specified subroutine, passing the token as an argument. This has the effect of invoking the given subroutine whenever a token appears in the database that matches the given specification. It can be used at system startup time for a module programmer to set up "demon" routines within the module. Example: "Execute that routine whenever you see a stop sign."

5.5. Architecture

Several modules use the CODGER tools and fit into a higher level architecture. The modules are:

- Pilot: Looks at the map and at current vehicle position to predict road location for Vision. Plans paths.
- Map Navigator: Maintains a world map, does global path planning, provides long-term direction to the Pilot. The world map may start out empty, or may include any level of detail up to exact locations and shapes of objects.
- Color Vision: Waits for a prediction from the Pilot, waits until the vehicle is in the best position to take an image of that section of the road, returns road location.
- Obstacle Detection: Gets a request from the Pilot to check a part of the road for obstacles. Returns a list of obstacles on or near that chunk of the road.
- Helm: Gets planned path from Pilot, converts polyline path into smooth arcs, steers vehicle.
- Graphics and Monitor: Draws or prints position of vehicle, obstacles, predicted and

perceived road.

There are two other modules in our architecture. These have not yet been implemented:

- Captain: Talks to the user and provides high-level route and mission constraints such as *avoid area A* or *go by road B*.
- Lookout: Looks for landmarks and objects of importance to the mission.

These modules use CODGER to pass information about *driving units*. A driving unit is a short chunk of the road or terrain (in our case 4 m long) treated as a unit for perception and path planning. The Pilot gives driving unit predictions to Color Vision, which returns an updated driving unit location. Obstacle Detection then sweeps a driving unit for obstacles. The Pilot takes the driving unit and obstacles, plans a path, and hands the path off to the Helm. The whole process is set up as a pipeline, in which Color Vision is looking ahead 3 driving units, Obstacle Detection is looking 2 driving units ahead, and path planning at the next unit. If for any reason some stage slows down, all following stages of the pipeline must wait. So, for instance, if Color Vision is waiting for the vehicle to come around a bend so it can see down the road, Obstacle Detection will finish its current unit and will then have to wait for Color Vision to proceed. In an extreme case, the vehicle may have to come to a halt until everything clears up. All planned paths include a deceleration to a stop at the end, so if no new path comes along to overwrite the current path the vehicle will stop before driving into an area that has not been seen or cleared of obstacles.

In our current system and test area, 3 driving units is too far ahead for Color Vision to look, so both Color Vision and Obstacle Detection are looking at the same driving unit. Obstacle Detection looks at an area sufficiently larger than the Pilot's predicted driving unit location to guarantee that the actual road is covered. Another practical modification is to have Obstacle Detection look at the closest driving unit also, so a person walking onto the road immediately in front of the vehicle will be noticed. Our system will try to plan a path around obstacles while remaining on the road. If that is not possible, it will come to a halt and wait for the obstacle to move before continuing.

6. Conclusions and Future Work

The system described here works. It has successfully driven the Navlab many tens of times, processing thousands of color and range images without running off the road or hitting any obstacles. CODGER has proved to be a useful tool, handling many of the details of communications and geometry. Module developers have been able to build and test their routines in isolation, with relatively little integration overhead. Yet there are several areas that need much more work.

Speed. We drive the Navlab at 10 cm/sec, a slow shuffle. Our slow speed is because our test road is narrow and winding, and because we deliberately concentrate on competence rather than on speed. But faster motion is always more interesting, so we are pursuing several ways of increasing speed. One bottleneck is the computing hardware. We are mounting a Warp, Carnegie Mellon's experimental high-speed processor, on the Navlab. The Warp will give us a factor of 100 more processing power than a Sun for color and range image processing. At the same time, we are looking at improvements in the software architecture. We need a more sophisticated path planner, and we need to process images that are more closely spaced than the length of a driving unit. Also, as the vehicle moves more quickly, our simplifying assumption that steering is instantaneous and that the vehicle moves along circular arcs becomes more seriously flawed. We are looking at other kinds of smooth arcs, such as clothoids. More

important, the controller is evolving to handle more of the low-level path smoothing and following.

Map. One reason for the slow speed is that the Pilot assumes straight roads. We need to have a description that allows for curved roads, with some constraints on maximum curvature. The next steps will include building maps as we go, so that subsequent runs over the same course can be faster and easier.

Cross-country travel. Travel on roads is only half the challenge. The Navlab should be able to leave roads and venture cross-country. Our plans call for a fully integrated on-road/off-road capability.

Intersections. Current vision routines have a built-in assumption that there is one road in the scene. When the Navlab comes to a fork in the road, vision will report one or the other of the forks as the true road depending on which looks bigger. It will be important to extend the vision geometry to handle intersections as well as straight roads. We already have this ability on our sidewalk system and will bring that over to the Navlab. Vision must also be able to find the road from offroad.

Landmarks. Especially as we venture off roads, it will become increasingly important to be able to update our position based on sighting landmarks. This involves map and perception enhancements, plus understanding how to share limited resources, such as the camera, between path finding and landmark searches.

Software Development. Our current blackboard system can manipulate primitive data elements but has no concept of data structures made up of tokens on the blackboard. We need aggregate data types for representing complex 3-D geometric descriptions of objects for recognition. We will also be implementing a Lisp interface to our blackboard. All current modules are written in C, but we will soon want to write higher-level modules in Lisp.

Integration with Work from Other Sites. Other universities and research groups cooperating with Carnegie Mellon through DARPA Strategic Computing Vision program. We plan to incorporate some of their programs into the Navlab system in the coming years as it evolves into the "new generation vision system" that is the goal of that program.

Acknowledgments

The Terregator and Navlab were built by William Whittaker's group in the Construction Robotics Laboratory, and the Warp group is led by H. T. Kung and Jon Webb. The real work gets done by an army of eight staff, nine graduate students, five visitors, and three part time programmers.

This research was supported by the Strategic Computing Initiative of the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5351, and monitored by the U.S. Army Engineer Topographic Laboratories under contract DACA76-85-C-0003. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

References

[Cohen 82] Cohen, P., Barr, A., Feigenbaum, E., eds.
The Handbook of Artificial Intelligence.
William Kaufman, 1982.

[Goto 86] Goto, Y., Matsuzaki, K., Kweon, I., Obatake, T.
CMU sidewalk navigation system.
In *Fall Joint Computer Conference.* ACM/IEEE, 1986.

[Hebert 86] Hebert, M., Kanade, T.
Outdoor scene analysis using range data.
In *IEEE International Conference on Robotics and Automation.* 1986.

[Rychener 86] Rychener, M. D., Farinacci, M. L., Hulthage, I., Fox, M. S.
Integration of multiple knowledge sources in Alladin, an alloy design system.
In *AAAI-1986.* AAAI, 1986.

[Shafer 86] Shafer, S., Stentz, A., Thorpe, C.
An architecture for sensor fusion in a mobile robot.
In *IEEE International Conference on Robotics and Automation.* 1986.

[Singh 86] Singh, J., et al.
NavLab: an autonomous vehicle.
Technical Report, Carnegie Mellon Robotics Institute, 1986.

[Thorpe 86] Thorpe, C.
Vision and navigation for the CMU Navlab.
In *SPIE.* Society of Photo-Optical Instrumentation Engineers, October, 1986.

Section IV

The CMU System for Mobile Robot Navigation

Yoshimasa Goto

Anthony Stentz

The Robotics Institute

Carnegie-Mellon University

Pittsburgh, PA 15213

Abstract

This paper describes the current status of the Autonomous Land Vehicle research at Carnegie Mellon University's Robotics Institute, focusing primarily on the system architecture. We begin with a discussion of the issues concerning outdoor navigation, then describe the various perception, planning, and control components of our system that address these issues. We describe the CODGER software system for integrating these components into a single system, synchronizing the data flow between them in order to maximize parallelism. Our system is able to drive a robot vehicle continuously with two sensors, a color camera and a laser rangefinder, on a network of sidewalks, up a bicycle slope, and through a curved road through an area populated with trees. Finally, we discuss the results of our experiments, as well as problems uncovered in the process and our plans for addressing them.[1]

1. Introduction

The goal of the Autonomous Land Vehicle group at Carnegie Mellon University is to create an autonomous mobile robot system capable of operating in outdoor environments. Because of the complexity of real-world domains and the requirement for continuous and real-time motion, such a robot system needs system architectural support for multiple sensors and parallel processing. These capabilities are not found in simpler robot systems. At CMU, we are studying mobile robot system architecture and have developed the navigation system working at two test sites and on two experimental vehicles [2, 3, 4, 8, 10, 11]. This paper describes the current status of our system and some problems uncovered through real experiments.

1.1. The Test Sites and Vehicles

We have two test sites, the Carnegie Mellon campus and an adjoining park, Schenley Park. The CMU campus test site has a sidewalk network including intersections, stairs and bicycle slopes (figure 1). The Schenley Park test site has curved sidewalks in an area well populated with trees (figure 2).

Figure 3 shows our two experimental vehicles, the Navigation Laboratory (Navlab) used in the

[1]This research was supported by the Strategic Computing Initiative of the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5351, and monitored by the U.S. Army Engineer Topographic Laboratories under contract DACA76-85-C-0003. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United State Government.

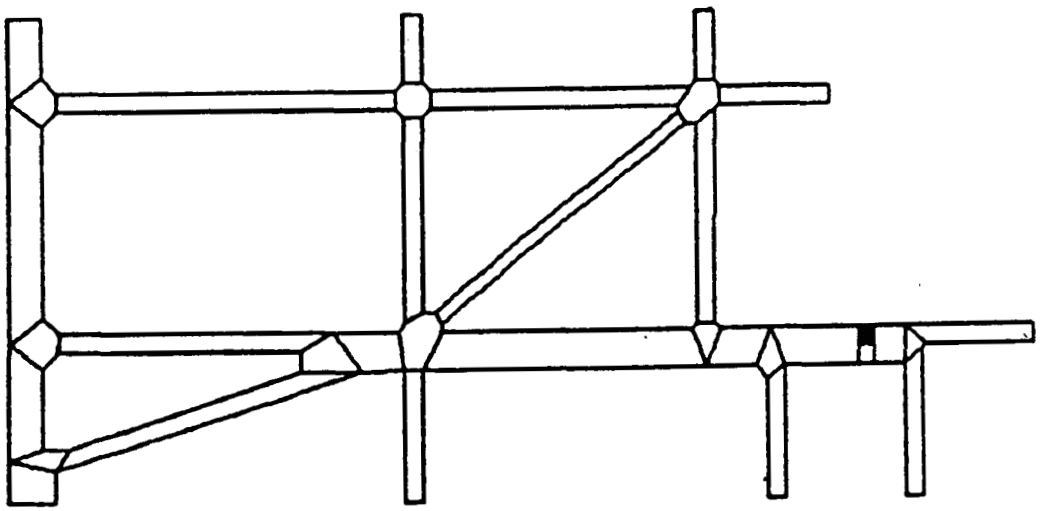


Figure 1: Map of the CMU Campus Test Site

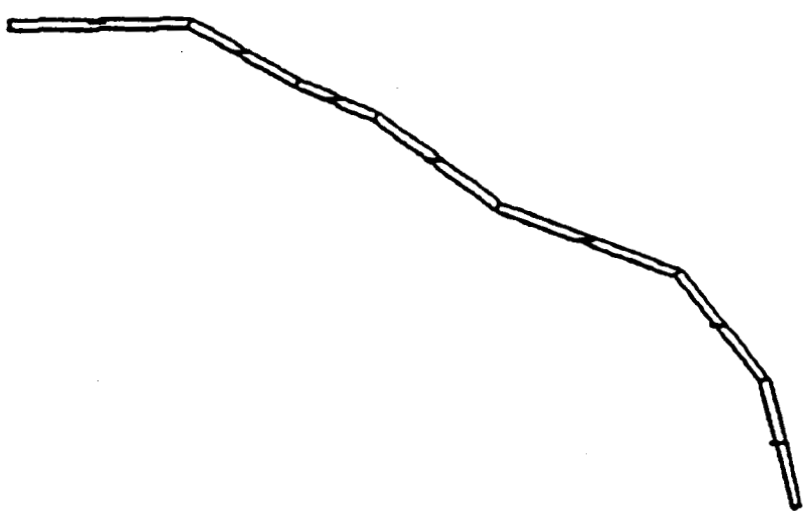


Figure 2: Map of the Schenley Park Test Site

Schenley Park test site, and the Terregator used in the CMU campus test site. Both of them are equipped with a color TV camera and a laser rangefinder made by ERIM. The Navlab carries four general purpose computers (SUN-3s) on board. The Terregator is linked to SUN-3s in the laboratory with radio communication. All of the SUN-3s are interconnected with a EtherNet. Our navigation system works on both vehicles in each test site.

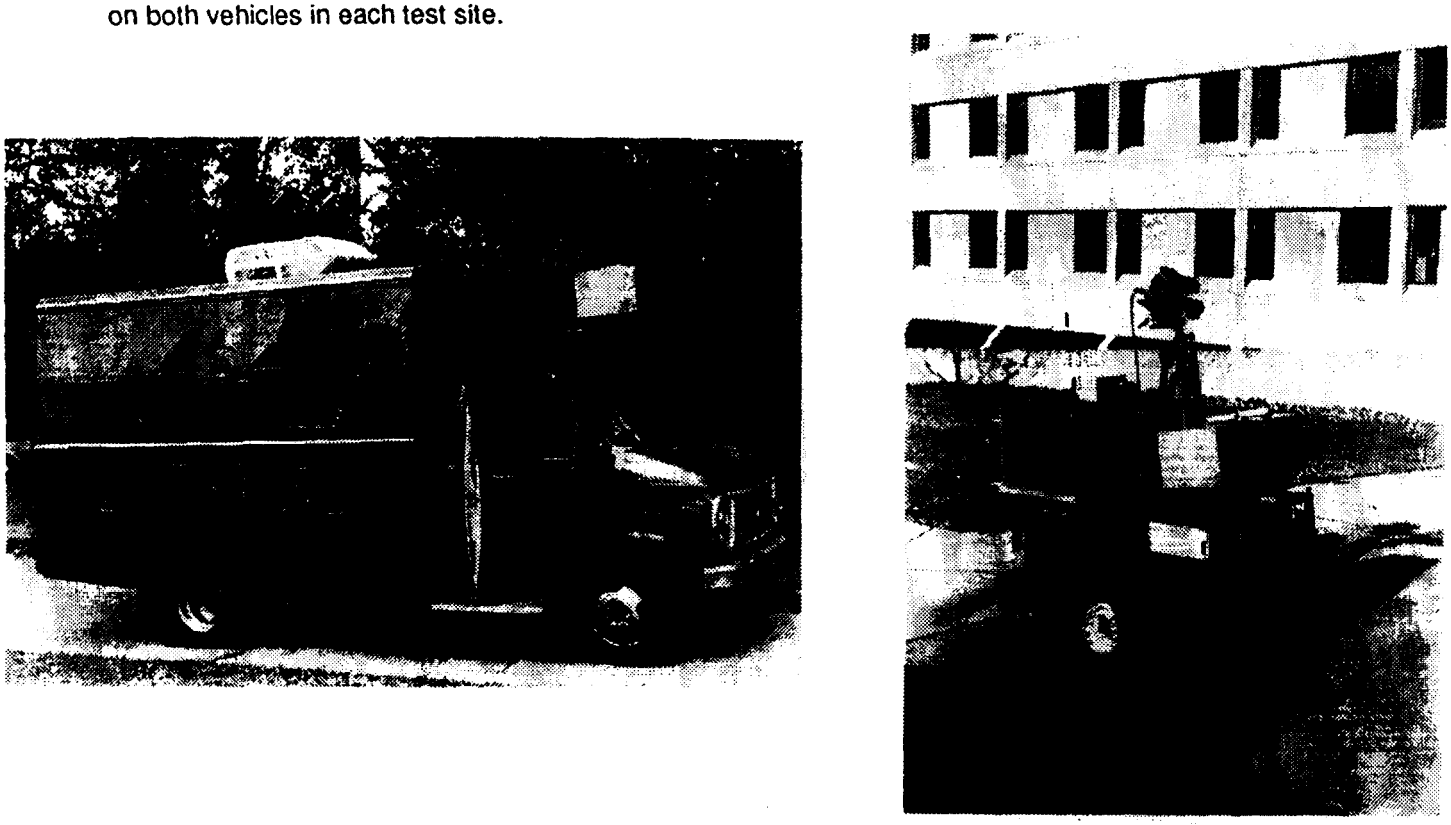


Figure 3: The Navlab and Terregator

1.2. Current System Capabilities

Currently, the system has the capability.

- to execute a prespecified user mission over a mapped network of sidewalks, including turning at the intersections and driving up the bicycle slope;
- to recognize landmarks, stairs and intersections;
- to drive on unmapped, curved, ill-defined roads using assumptions about local road linearity;
- to detect obstacles and stop until they move away;
- to avoid obstacles; and
- to drive continuously at 200mm/sec.

2. Design of the System Architecture

In this section we describe the goals of our outdoor navigation system and the design principles, followed by an analysis of the outdoor navigation task itself. We describe our system architecture as it is shaped by these principles and analyses.

2.1. Design Goals and Principles

The goals of our outdoor navigation system are:

- **map-driven mission execution:** The system drives the vehicle to reach a given goal position.
- **on- and off-road navigation:** Navigation environments include not only roads but also open terrain.
- **landmark recognition:** Landmark sightings are essential in order to correct for drift in the vehicle's dead-reckoning system.
- **obstacle avoidance**
- **continuous motion in real time:** Stop and go motion is unacceptable for our purposes. Perception, planning, and control should be carried out while the vehicle is moving at a reasonable speed.

In order to satisfy these goals, we have adopted the following design principles.

- **sensor fusion:** A single sensor is not enough to analyze complex outdoor environments. Sensors include not only a TV camera and a range sensor but also an inertial navigation sensor, a wheel rotation counter, etc.
- **parallel execution:** In order to process data from a number of sensors, make global and local plans , and drive the vehicle in real-time, parallelism is essential.
- **flexibility and extensibility:** This principle is essential because the whole system is quite large, requiring the integration of a wide range of modules.

2.2. Outdoor Navigation Tasks

Outdoor navigation includes several different navigation modes. Figure 4 illustrates several examples. On-road vs. off-road is just one example. Even in on-road navigation, *turning at the intersection* requires more sophisticated driving skill than *following the road.* In road following, the assumption that the ground is flat makes perception easier, but *driving through the forest* does not satisfy this assumption and requires more complex perception processing.

According to this analysis we decompose outdoor navigation into two navigation levels: *global* and *local.* At the global level, the system tasks are to select the best navigation route to reach the destination given by a user mission, and to divide the whole route into a sequence of *route segments*, each corresponding to a uniform driving mode. The current system supports the following navigation modes: *following the road, turning at the intersection, driving up the slope.*

Local navigation involves driving within a single route segment. The navigation mode is uniform and the system drives the vehicle along the route segment continuously, perceiving objects, planning path plans, and controlling the vehicle. The important thing is that these tasks, perception, planning, and control, form a cycle and can be executed concurrently.

2.3. System Architecture

Figure 5 is a block diagram of our system architecture. The architecture consists of several modules and a communications database which links the modules together.

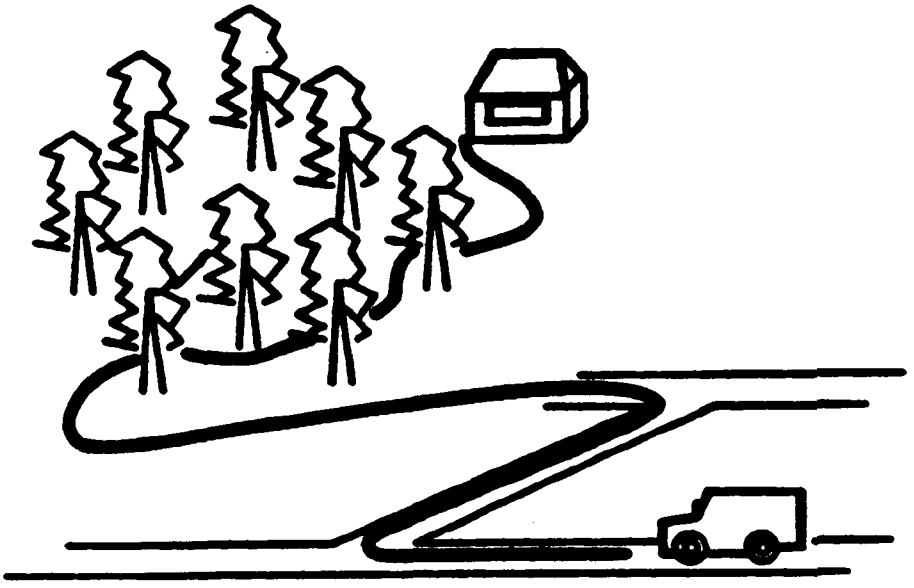


Figure 4: Outdoor navigation

2.3.1. Module Structure

In order to support the tasks described in the previous section, we first decomposed the whole system into the following modules:

- **CAPTAIN** executes user mission commands and sends the destination and the constraints of each mission step to the MAP NAVIGATOR one step at a time, and gets the result of each mission step.
- **MAP NAVIGATOR** selects the best route by searching the Map Database, decomposes it into a sequence of route segments, generates a route segment description which includes objects from the Map visible from the route segment, and sends it to the PILOT.
- **PILOT** coordinates the activities of PERCEPTION and the HELM to perform local navigation continuously within a single route segment.
- **PERCEPTION** uses sensors to find objects predicted to lie within the vehicle's field of view. It estimates the vehicle's position if possible.
- **HELM** gets the local path plan generated by the PILOT and drives the vehicle.

The PILOT is decomposed into several submodules which run concurrently (figure 6).

- **DRIVING MONITOR** decomposes the route segment into small pieces called *driving units*. A driving unit is the basic unit for perception, planning, and control processing at the local navigation level. For example, PERCEPTION must be able to process a whole driving unit with a single image. The DRIVING MONITOR creates a *driving unit description* , which describes objects in the driving unit, and sends it to the following submodules.
- **DRIVING UNIT FINDER** functions as an interface to PERCEPTION, sending the driving unit description to it and getting the result from it.

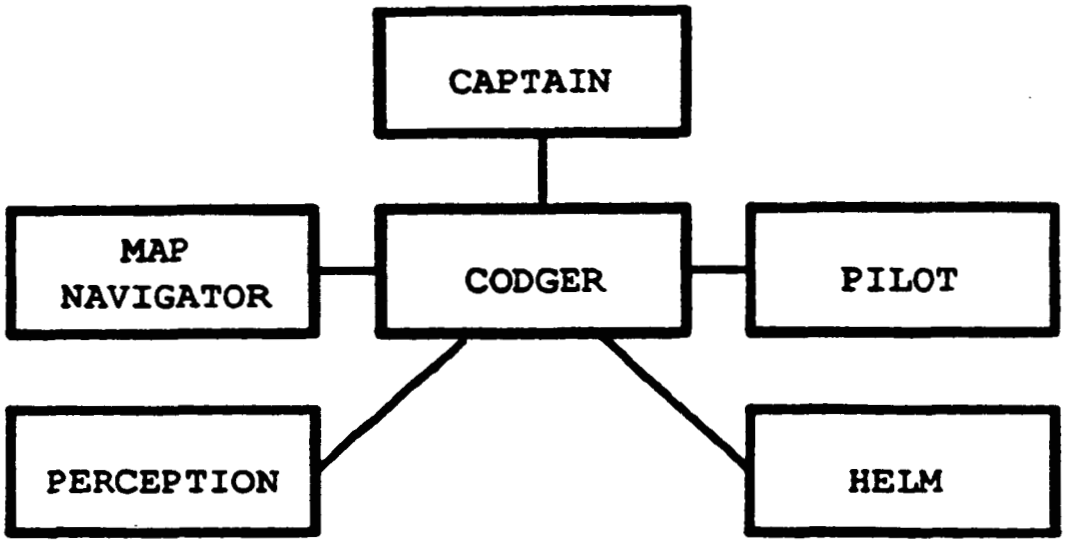


Figure 5: System architecture

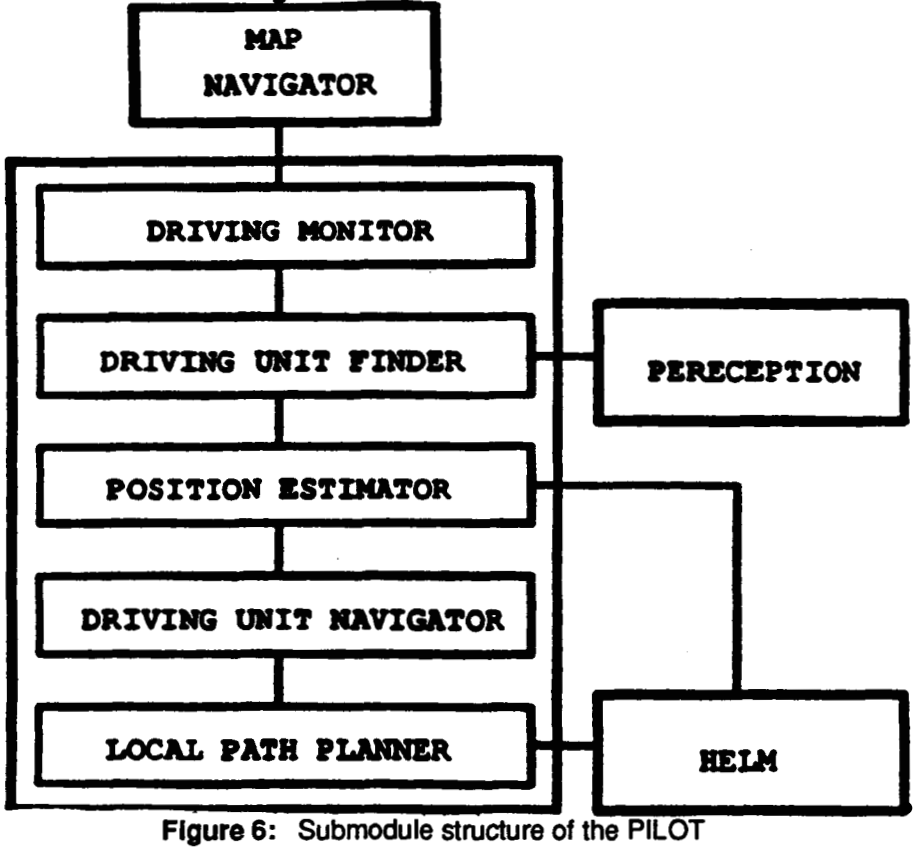


Figure 6: Submodule structure of the PILOT

- **POSITION ESTIMATOR** estimates the vehicle position using both the result of PERCEPTION and dead-reckoning.
- **DRIVING UNIT NAVIGATOR** determines the admissible passage in which to drive the vehicle.
- **LOCAL PATH PLANNER** generates the path plan within the driving unit, avoids obstacles and keeps the vehicle in the admissible passage. The path plan is sent to the HELM.

2.3.2. CODGER

It is important not only to build the modules, but also to connect them into a coherent system. Based on our design principles, we have created a software system called *CODGER* (COmmunications Database with GEometric Reasoning) which supports parallel asynchronous execution and communication between the modules. We describe CODGER in detail in the next section.

3. Parallelism

3.1. The CODGER System for Parallel Processing

In order to navigate in real-time, we have employed parallelism in our perception, planning, and control subsystems. Our computing resources consist of several SUN-3 microcomputers, VAX minicomputers, and a high-speed, parallel processor known as the WARP interconnected with an EtherNet. We have designed and implemented a software system called CODGER (COmmunications Database with GEometric Reasoning) [9] to effectively utilize this parallelism.

The CODGER system consists of a central database (*Local Map*), a process that manages this database (*Local Map Builder or LMB*), and a library of functions for accessing the data (*LMB interface*) (see Figure 7). The various perceptual, planning, and control modules in the system are compiled with the LMB interface and invoke functions to store and retrieve data from the central database. The CODGER system can be run on any mix of SUN-3s and VAXes and handles data type conversions automatically. This system permits highly modular development requiring recompilation only for modules directly affected by a change.

3.1.1. Data Representation

Data in the Local Map is represented in *tokens* consisting of lists of *attribute-value* pairs. Tokens can be used to represent any information including physical objects, hypotheses, plans, commands, and reports. The token types are defined in a *template file* which is read by the LMB at system startup time. Attribute types may be the usual scalars (e.g., floats, integers), sets of scalars, or geometric locations. Geometric locations consist of a two- dimensional, polygonal *shape* and a reference coordinate *frame*. The CODGER system provides mechanisms for defining coordinate frames and for automatically converting geometric data from one frame to another, thereby allowing modules to retrieve data from the database and representing it in a form meaningful to them. Geometric data is the only data interpreted by the CODGER system; the interpretation of all other data types is delegated to the modules that use them.

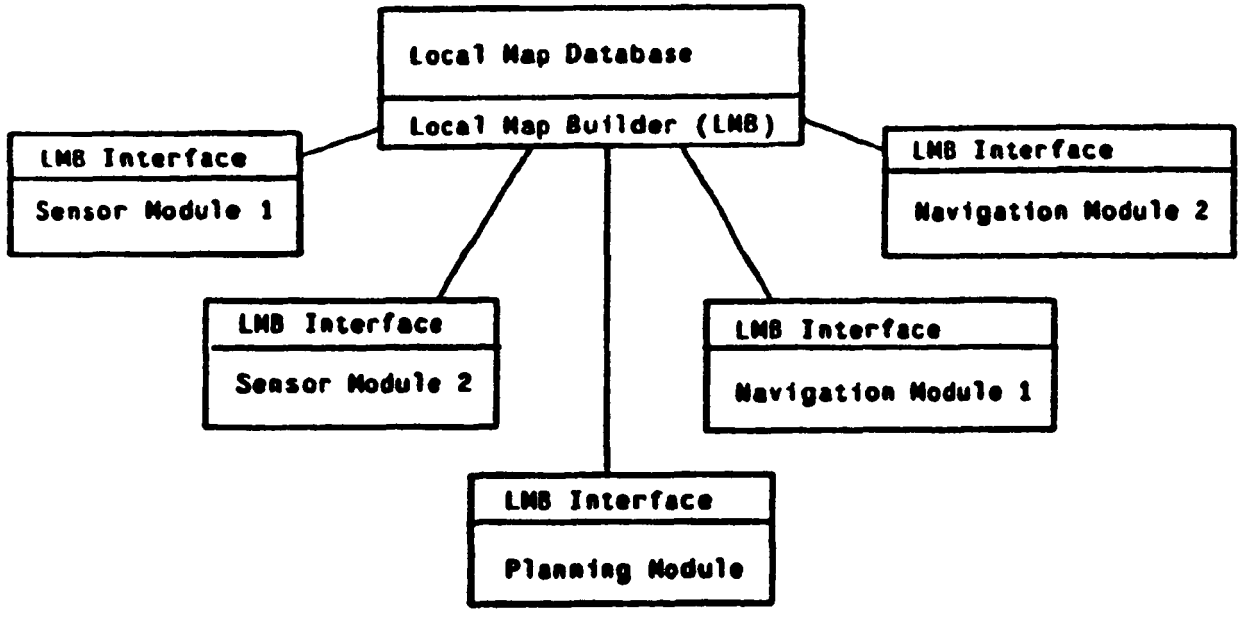


Figure 7: The CODGER software system

3.1.2. Synchronization

The LMB interface provides functions for storing and retrieving data from the central database. Tokens can be retrieved using *specifications*. Specifications are simply boolean expressions evaluated across token attribute values. A specification may include computations such as mathematical expressions, boolean relations, and comparisons between attribute values. Geometric indexing is of particular importance for a mobile robot system. For example, the planner needs to search a database of map objects to locate suitable landmarks or to find the shortest path to the goal. The CODGER system provides a host of functions including those for computing the distance and intersection of locations. These functions can be embedded in specifications and matched to the database.

The CODGER system has a set of primitives to ensure that data transfer between system modules is synchronized and runs smoothly. The synchronization is implemented in the data retrieval mechanism. Specifications are sent to the LMB as either one-shot or standing requests. For one-shot specs, the calling module blocks while the LMB matches the spec to the tokens. Tokens that match are retrieved and the module resumes execution. If no tokens match, either the module stays blocked until a matching token appears in the database or an error is returned and the module resumes execution, depending on an option specified in the request. For example, the PATH PLANNER may use a one-shot to find obstacles stored in the database *before* it can plan a path. In contrast, the HELM, which controls the vehicle, uses a standing spec to retrieve tokens supplying steering commands *whenever* they appear.

3.2. Parallel Asynchronous Execution of Modules

Thus far we have run our scenarios with four SUN-3s interconnected with an EtherNet. The CAPTAIN, MAP NAVIGATOR, PILOT, and HELM are separate modules in the system, and PERCEPTION is two modules (range and camera image processing). All of the modules run in parallel; they synchronize themselves through the LMB database.

3.2.1. Global and Local Navigation

A good example of parallelism in the system is the interaction between the CAPTAIN, MAP NAVIGATOR, and PILOT. The CAPTAIN and MAP NAVIGATOR search the map database to plan a global path for the vehicle in accordance with the mission specification. The PILOT coordinates PERCEPTION, PATH PLANNING, and control through the HELM to navigate locally. The global and local navigation operations run in parallel. The MAP NAVIGATOR monitors the progress of the PILOT to ensure that the PILOT's transition from one route segment to the next occurs smoothly.

3.2.2. Driving Pipeline

Another good example of parallelism is within the PILOT itself. As described earlier, the PILOT monitors local navigation. For each driving unit, the PILOT performs four operations in the following order: predict it, recognize with the camera and scan it for obstacles with the rangefinder, establish driving constraints and plan a path through it, and oversee the vehicle's execution of it. In the PILOT, these four operations are separate modules linked together in a pipeline (see Figure 8). While in steady state, the PILOT is predicting a driving unit 12 to 16 meters in front of the vehicle, recognizing a driving unit and scanning it for obstacles (in parallel) 8 to 12 meters in front, planning a path 4 to 8 meters in front, and driving to a point 4 meters in front. The stages of the pipeline synchronize themselves through the CODGER database.

The processing times for each stage vary as a function of the navigation task. In navigation on uncluttered roads, the vision subsystem requires about 10 seconds of real-time per image, the range subsystem requires about 6 seconds, and the local path planner requires less than a second. In this case, the stage time of the pipeline is that of the vision subsystem: 10 seconds. In cluttered environments, the local path planner may require 10 to 20 seconds or more, thereby becoming the bottleneck. In either case, the vehicle is not permitted to drive on to a driving unit until it has propagated through all stages of the pipeline (i.e., all operations have been performed on it). For example, when driving around the corner of a building, the vision stage must wait until the vehicle reaches the corner in order to see the next driving unit. Once the vehicle reaches the corner, it must stop while waiting for the vision, scanning, and planning stages to process the driving unit before driving again.

4. Sensor Fusion

4.1. Types of Sensor Fusion

The Navlab and Terregator vehicles are equipped with a host of sensors including color cameras, a laser rangefinder, and motion sensors such as a gyro and shaft-encoder counter. In order to obtain a single, consistent interpretation of the vehicle's environment, the results of these sensors must be fused. We have identified three types of sensor fusion [8]:

- **Competitive:** Sensors provide data that either agrees or conflicts. This case arises when

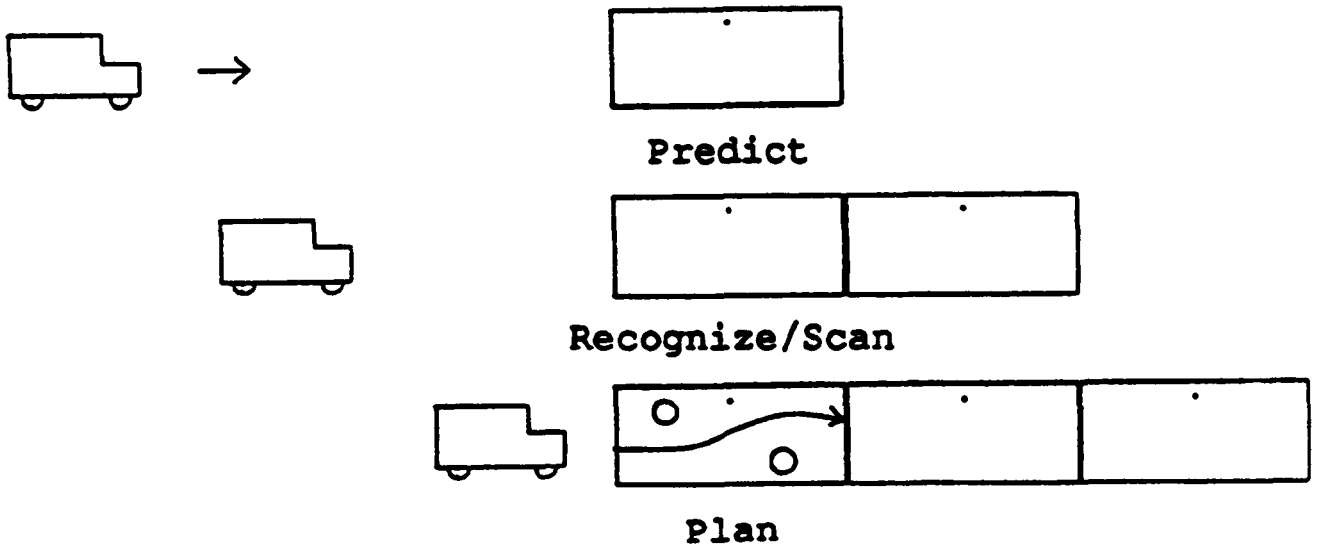


Figure 8: Driving pipeline

sensors provide data of the same modality. In the CMU systems, the task of determining the vehicle's position best characterizes this type of fusion. Readings from the vehicle's dead-reckoning system as well as landmark sightings provide estimates of the vehicle's position.

- **Complementary:** Sensors provide data of different modalities. The task of recognizing three-dimensional objects illustrates this kind of fusion. In the CMU systems, a set of stairs is recognized using a color camera and laser rangefinder. The color camera provides *image* information (e.g., color and texture) while the laser rangefinder provides *three-dimensional* information.
- **Independent:** A single sensor is used for each task. An example of a task requiring a single sensor is distant landmark recognition. In this case, only the camera is used for landmarks beyond the range of the laser rangefinder.

4.2. Examples of Sensor Fusion Tasks

4.2.1. Vehicle Position Estimation

In our road following scenarios, vehicle position estimation has been the most important sensor fusion task. By vehicle position, we mean the position and orientation of the vehicle in the ground plane (3 degrees of freedom) relative to the world coordinate frame. In the current system, there are two sources of position information. First, dead-reckoning provides vehicle-based position information. The CODGER system maintains a history of the steering commands issued to the vehicle, effectively recording the trajectory of the vehicle from its starting point.

Second, landmark sightings directly pinpoint the position of the vehicle with respect to the world at a point in time. In the campus test site, the system has access to a complete topographical map of the

sidewalks and intersections on which it drives. The system uses a color camera to sight the intersections and sidewalks and uses these sightings to correct the estimate of the vehicle's position. The intersections are of rank three, meaning that the position and orientation of the vehicle with respect to the intersection can be determined fully (to three degrees of freedom) from the sighting. Our tests have shown that such landmark sightings are far more accurate but less reliable than the current dead-reckoning system, that is, landmark sightings provide more accurate vehicle position estimates; however, the sightings occasionally fail. If the vehicle position estimates from the sighting and dead-reckoning disagree drastically, the *conflict* is settled in favor of the dead-reckoning system; otherwise, the result from the landmark sighting is used. In this case, the CODGER system adjusts its record of the vehicle's trajectory so that it agrees with the most recent landmark sighting, and discards all previous sightings.

The CODGER system is able to handle landmark sightings of rank less than three. The most common "landmark" in our scenarios is the sidewalk on which the vehicle drives. Since a sidewalk sighting provides only the orientation and perpendicular distance of the vehicle with respect to the sidewalk, the correction is of rank two. Therefore, the position of the vehicle is constrained to lie on a straight line. The CODGER system projects the position of the vehicle from dead-reckoning onto this line and uses the projected point as a full (rank three) correction. Since most of the error in the vehicle's motion is lateral drift from the road, this approximation works well.

4.2.2. Pilot Control

Complementary fusion is grounded in the Pilot's control functions. The Pilot ensures that the vehicle travels only where it is permitted and where it can. For example, the color camera is used to segment road from nonroad surfaces. The laser rangefinder scans the area in front of the vehicle for obstacles or unnavigable (i.e., rough or steep) terrain. The road surface is fused with the free space and is passed to the local path planner. Since the two sensor operations do not necessarily occur at the same time, the vehicle's dead-reckoning system also comes into play.

4.2.3. Colored Range Image

Another example of complementary fusion of camera and range data is the colored range image. A colored range image is created by "painting" a color image onto the depth map of a range image. The resultant image is used in our systems to recognize complicated three dimensional objects such as a set of stairs. In order to avoid the relatively large error in the vehicle's dead-reckoning system, the vehicle remains motionless while digitizing a corresponding pair of camera and range images [2].

4.3. Problems and Future Work

We have plans for improving our sensor fusion mechanisms. Currently, the CODGER system handles competing sensor data by retaining the most recent measurement and discarding all others. This is undesirable for the following reasons. First, a single bad measurement (e.g., landmark sighting) can easily throw the vehicle off track. Second, measurements can reinforce each other. By discarding old measurements, useful information is lost. A weighting scheme is needed for combining competing sensor data. In many cases, it is useful to model error in sensor data as gaussian noise. For example, error in dead-reckoning may arise from random error in the wheel velocities. Likewise, quantization error in range and camera images can be modeled as gaussian noise. A number of schemes exist for fusing such data ranging from simple Kalman filtering techniques to full-blown Bayesian observation networks [1] [7].

5. Local Control

In this section we discuss some of the control problems in local navigation.

5.1. Adaptive Driving Units and Sensor View Frames

Management of driving units and sensor view frames is essential in local control. As described in section 2, the driving unit is a minimum control unit, a unit to perceive objects, generate a path plan, and drive the vehicle. The PERCEPTION module digitizes an image in each driving unit, and the vehicle's position is estimated and its trajectory is planned once in each driving unit. Therefore, an appropriate driving unit size is essential for stable control. For example, the sensor view frame cannot cover a very large driving unit. Conversely, small driving units place rigid constraints on the LOCAL PATH PLANNER, because of the short distance between the starting point and the goal point. The aiming of the sensor view frame determines the point at which to digitize an image and to update the vehicle position and path plan.

In the current system, the sensor view frame is always fixed with respect to the vehicle. The size of the driving unit is fixed for driving on roads (4~6 meters length), and is changed for turning at intersections so that the entire intersection can be see in a single image and to increase driving stability (see Figure 9). This method works well in almost all situations in our current test site.

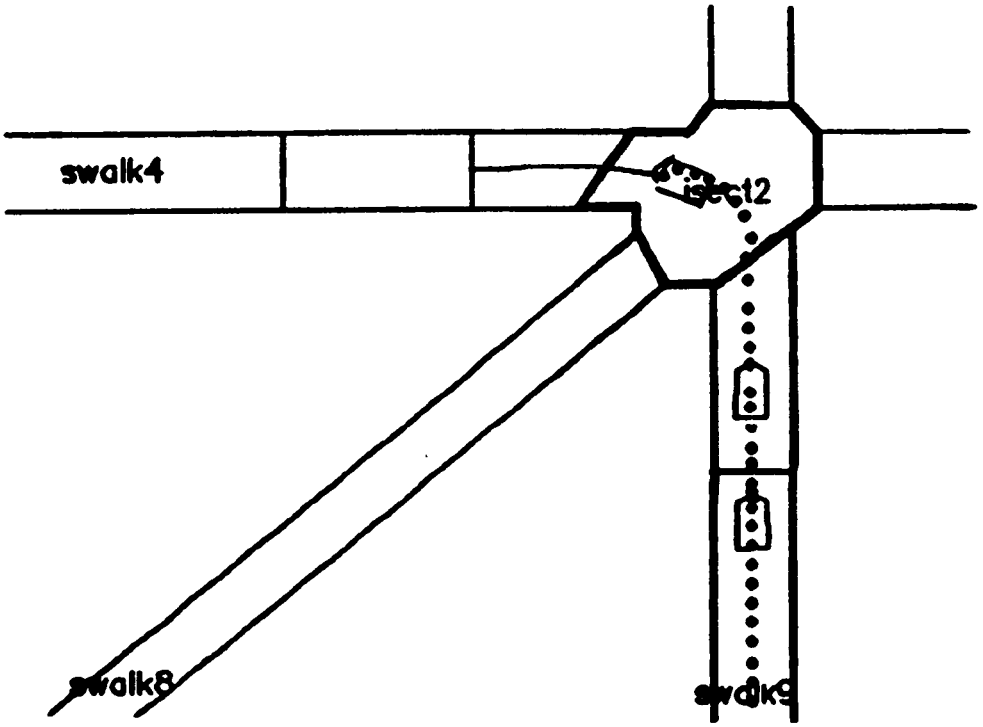


Figure 9: Intersection driving unit

For intersections requiring sharp turns (about 135 degrees), the current method does not suffice. Because there is only one driving unit at the intersection, the system digitizes an image, estimates the vehicle's position, and generates a path plan only once for a large turn. Furthermore, since the camera's field of view is fixed straight ahead, the system cannot see the driving unit after the intersection until the

vehicle has turned through the intersection. Though the actual path generated is not so bad, it is potentially unstable.

This experimental result indicates that the system should scan for an admissible passage, and update vehicle position estimation and local path plan more frequently when the vehicle changes its course faster. We plan to improve our method for managing driving units. Our new idea is:

- **length of the driving unit:** The length of the driving unit is bounded at the low end by the LOCAL PATH PLANNER's requirements for generating a reasonable path plan, and at the high end by the view frame required by PERCEPTION for recognizing a given object.
- **Driving unit interval:** The *driving unit interval* is the distance between the centers of adjacent driving units. Adjacent driving units can be overlapped, that is, they can be placed such that their interval is shorter than their length. Figure 10 illustrates this situation.

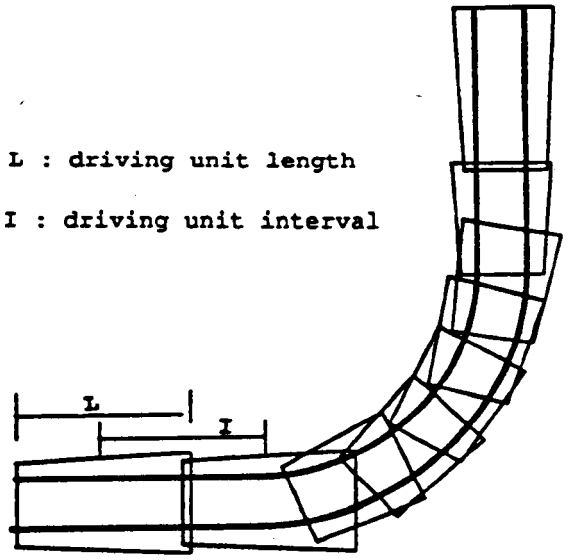


Figure 10: Adaptive Driving Units

- **Adjusting size and interval of driving unit:** If the passage is simple, the length and interval of the driving unit is long. If the passage is complex, for example, in the case of highly curved roads or intersections, or in the presence of obstacles, the length and interval of driving unit are shorter. And if the required driving unit interval must be shorter than the length of driving unit, the driving units are overlapped. Therefore, the vehicle's position is estimated and a local path is planned more frequently so that the vehicle drives stably (figure 10).
- **Adjusting sensor view frame:** The sensor view frame with respect to the vehicle, the distance and the direction to the driving unit from the vehicle, is adjusted using the *pan and tilt* mechanism of the sensor. In most cases, a longer distance to the next driving unit allows a higher vehicle speed. If the processing time of the PERCEPTION and the PILOT is constant, the longer distance means a higher vehicle speed. But the longer distance produces less accuracy in perception and vehicle position estimation. Therefore, the distance is determined for the required accuracy, which depends on the complexity of

passage. Using the pan and tilt mechanism, PERCEPTION can digitize an image at the best distance from the driving unit, since the sensor's view frame is less rigidly tied to the orientation and position of the vehicle.

5.2. Vehicle Speed

It is an important capability of an autonomous mobile robot to adjust the vehicle's speed automatically so that the vehicle drives safely at the highest possible speed. The current system slows the vehicle down in turning to reduce driving error.

The delay in processing in the LOCAL PATH PLANNER and in communication between the HELM and the actual vehicle mechanism gives rise to errors in vehicle position estimation. For example, because of continuous motion and non-zero processing time, the vehicle position used by the LOCAL PATH PLANNER as a starting point differs slightly from the vehicle position when the vehicle starts executing the plan. Because the smaller turning radii give rise to larger errors in the vehicle's heading, which are more serious than displacement errors, the HELM slows the vehicle for turns with smaller radii. This method is useful for making the vehicle motion stable.

We will add to the system the capability for adjusting the vehicle speed to the highest possible value automatically. Our idea is the following:

- **schedule token:** The modules and the submodules working at the local navigation level store their predicted processing times in a *schedule token* in each cycle. PERCEPTION is the most time consuming module, and its processing time varies drastically from task to task.
- **adjusting vehicle speed:** Using the path plan and the predicted processing time stored in the schedule token, the HELM calculates and adjusts vehicle speed so that the speed is maximum and the modules can finish processing the driving unit before the vehicle reaches the end of the current planned trajectory.

5.3. Local Path Planning and Obstacle Avoidance

Local path planning is the task of finding a trajectory for the vehicle through admissible space to a goal point. In our system, the vehicle is constrained to move in the ground plane around obstacles (represented by polygons) while remaining within the driving unit (also a polygon). We have employed a configuration space approach [5] [6]. This algorithm, however, assumes that the vehicle is omnidirectional. Since our vehicles are not, we smooth the resultant path to ensure that the vehicle can execute it. The smoothed path is not guaranteed to miss obstacles. We plan to overcome this problem by developing a path planner that reasons about constraints on the vehicle's motion.

6. Navigation Map

Some information about the vehicle's environment must be supplied to the system a priori, even if it is incomplete, and even if it is nothing more than a data format for storing explored terrain. The user mission, for example, "turn at the second cross intersection and stop in front of the three oak trees" does not make sense to the system without a description of the environment. *The Navigation Map* is a data base to store the environment description needed for navigation.

6.1. Map Structure

The navigation map is a set of descriptions of physical objects in the navigation world. It is composed of two parts, the geographical map and the object data base. The geographical map stores object locations with their contour polylines. The object data base stores object geometrical shapes and other attributes, for example, the navigation cost of objects. Though, in the current system, all objects are described with both the geographical map and the object data base, in general, either of them can be unused. For example, the location of *stairs A* is known, but its shape is unknown.

The shape description is composed of two layers. The first layer stores shape attributes. For example, the width of the road, the length of the road, the height of the stairs , the number of steps, etc. The second layer stores actual geometrical shapes represented by the surface description. It is easy to describe incomplete shape information with only the first layer.

6.2. Data retrieval

The map data is stored in the CODGER data base as a set of tokens forming a tree structure. In order to retrieve map data, parent tokens have indexes to child tokens. Because the current CODGER system provides modules with a token retrieval mechanism that can pick up only one token at a time, retrieving large portions of the map is cumbersome. We plan to extend CODGER so that it can match and retrieve larger structures, possibly combined with an inheritance mechanism.

7. Other Tasks of the System

Navigation is just one goal of a *mobile robot system*. Generally speaking, however, navigation itself is not an end, but actually a means to achieve the final goals of the autonomous mobile robot system, such as carrying baggage, exploration, or refueling. Therefore, the system architecture must be able to accommodate tasks other than navigation.

Figure 11 illustrates one example of an extended system architecture which loads, carries and unloads baggage. The whole system is comprised of four layers, *mission control*, *vehicle resource management*, *signal processing*, and *physical hardware*. The CAPTAIN, only one module in the mission control layer, stores the user mission steps, sends them to the vehicle resource management layer one by one, and oversees their execution.

In the vehicle resource management layer, there are different modules working for different tasks. Although their tasks are different, they all work in a symbolic domain and do not handle the physical world directly. These modules oversee mission execution, generate plans, and pass information to modules in the signal processing layer. Through CODGER, they can communicate with each other, if necessary. The MAP NAVIGATOR and the PILOT, parts of the navigation system, are included in the vehicle resource management layer. The MANIPULATOR makes a plan (e.g., how to load and unload baggage with the arm) and sends it to the ARM CONTROLLER.

The modules in the signal processing layer interact with the physical world using senors and actuators. For example, PERCEPTION processes signals from sensors, the HELM drives the physical vehicle, and the ARM CONTROLLER operates the robot arm. The bottom level contains the real hardware, even if it includes some primitive controller. The sensors, the physical vehicle, and the robot arm are included in this layer.

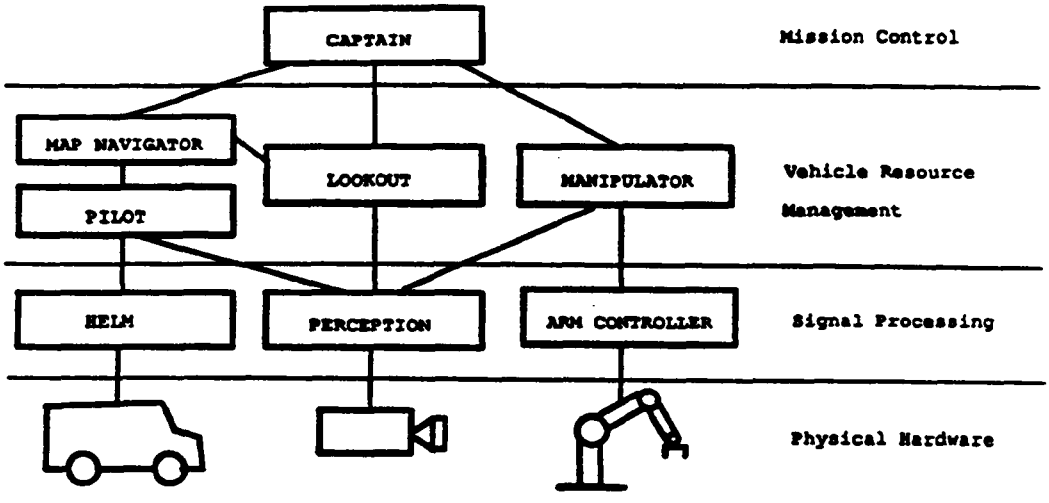


Figure 11: Extended system architecture

Because our current system architecture is built on the CODGER system it will be easy to expand to include these additional capabilities.

8. Conclusions

In this paper, we have described the CMU architecture for autonomous outdoor navigation. The system is highly modular and includes components for both global and local navigation. Global navigation is carried out by a route planner that searches a map database to find the best path satisfying a mission and oversees its execution. Local navigation is carried out by modules that use a color camera and a laser rangefinder to recognize roads and landmarks, scan for obstacles, reason about geometry to plan paths, and oversee the vehicle's execution of a planned trajectory.

The perception, planning, and control components are integrated into a single system through the CODGER software system. CODGER provides a common data representation scheme for all modules in the system with special attention paid to geometry. CODGER also provides primitives for synchronizing the modules in a way that maximizes parallelism at both the local and global levels.

We have demonstrated our system's ability to drive around a network of sidewalks and along a curved road, recognize complicated landmarks, and avoid obstacles. Future work will focus on improving CODGER for handling more difficult sensor fusion problems. We will also work on better schemes for local navigation and will strive to reduce our dependence on map data.

9. Acknowledgements

The design of our architecture was shaped by contributions from the entire Autonomous Land Vehicle group at CMU. We extend special thanks to Steve Shafer, Chuck Thorpe, and Takeo Kanade.

References

[1] Durrant-Whyte, H.
Integration, Coordination and Control of Multi-Sensor Robot Systems.
PhD thesis, University of Pennsylvania, 1986.

[2] Goto, Y., Matsuzaki, K., Kweon, I., Obatake, T.
CMU Sidewalk Navigation System.
In *FJCC-86.* 1986.

[3] Hebert, M. and Kanade, T.
Outdoor Scene Analysis Using Range Data.
In *Proc. 1986 IEEE Conference on Robotics and Automation.* April, 1986.

[4] Kanade, T., Thorpe, C., and Whittaker, W.
Autonomous Land Vehicle Project at CMU.
In *Proc. 1986 ACM Computer Conference.* Cincinnati, February, 1986.

[5] Lozano-Perez, T., Wesley, M. A.
An Algorithm for Planning Collison-Free Paths Among Polyhedral Obstacles.
Communications of the ACM 22(10), October, 1979.

[6] Lozano-Perez, T.
Spatial Planning: A Configuration Space Approach.
IEEE Transactions on Computers C-32(2), February, 1983.

[7] Mikhail, E. M., Ackerman, F.
Observations and Least Squares.
University Press of America, 1976.

[8] Shafer, S., Stentz, A., Thorpe, C.
An Architecture for Sensor Fusion in a Mobile Robot.
In *Proc. IEEE International Conference on Robotics and Automation.* April, 1986.

[9] Stentz, A., Shafer, S.
Module Programmer's Guide to Local Map Builder for NAVLAB.
1986.
In Preparation.

[10] Wallace, R., Stentz, A., Thorpe, C., Moravec, H., Whittaker, W., Kanade, T.
First Results in Robot Road-Following.
In *Proc. IJCAI-85.* August, 1985.

[11] Wallace, R., Matsuzaki, K., Goto, Y., Webb, J., Crisman, J., Kanade, T.
Progress in Robot Road Following.
In *Proc. IEEE International Conference on Robotics and Automation.* April, 1986.