First Year End Report for Perception for Outdoor Navigation

Charles Thorpe and Takeo Kanade

CMU-RI-TR-90-23

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

November 1990

Research supported by DARPA, DOD, monitored by the US Army Engineer Topographic Laboratories under contract DACA 76-89-C-0014, titled "Perception for Outdoor Navigation".

Table of Contents

3 Annotated Maps for Autonomous Land Vehicles

4 The Warp Machine on NAVLAB

5 YARF: A Progress Report

List of Figures

ANNOTATED MAPS

WARP ON NAVLAB

YARF

Chapter 1: Introduction

1.1 Abstract

This report reviews progress at Carnegie Mellon from August 16, 1989 to August 15, 1990 on research sponsored by DARPA, DOD, monitored by the US Army Engineer Topographic Laboratories under contract DACA 76-89-C-0014, titled "Perception for Outdoor Navigation".

Research supported by this contract includes perception for road following, terrain mapping for off-road navigation, and systems software for building integrated mobile robots. We overview our efforts for the year, and list our publications and personnel, then provide further detail on several of our subprojects.

1.2 Introduction

This report reviews progress at Carnegie Mellon from August 16, 1989 to August 15, 1990 on research sponsored by DARPA, DOD, monitored by the US Army Engineer Topographic Laboratories under contract DACA 76-89-C-0014, titled "Perception for Outdoor Navigation".

During the past year, this contract has supported research on color vision for road following; 3-D perception for terrain mapping and cross-country mobility; and system building for autonomous navigation. We have demonstrated autonomous navigation on a variety of roads, including single lane dirt, gravel, and paved; and multi-lane roads with and without lane markings. Our perception modules use a variety of techniques for video processing (clustering theory, symbolic feature detection, neural nets), and for range data analysis (landmark navigation, reflectance processing). We have also integrated position-based navigation (INS and GPS), and combinations of all these techniques into mobile robot systems and demonstrations. Our scientific papers this year include a book (Vision and Navigation: the CMU Navlab), three PhD dissertations, and an MS thesis.

In the first chapter of this report, we briefly summarize our progress over the past year, and list personnel and publications. Following chapters go into greater detail on individual projects and accomplishments.

1.2.1 SCARF

SCARF, which stands for Supervised Classification Applied to Road Following, tracks roads by adaptive color classification. Under previous contracts, we developed SCARF, implemented it on various computers including the Warp, and used it for many Navlab runs. During this past year, Crisman finished her thesis on SCARF [3]. This entailed three new research initiatives: intersection detection, analysis of performance on Warp, and tests on more road types.

SCARF operates in unstructured environments, where intersections may not be accurately mapped, and where the size and shape of branch roads may not be known. While our previous experiments with SCARF intersection detection used a template of the intersection shape, we wanted to build a system that did not use such strong models. Naive approaches to intersection detection could be computationally inefficient: an intersection with three roads, each of which has two degrees of freedom, could create a 6-D search. Our approach, instead, takes advantage of scene structure to decompose the search into several smaller searches. After we have classified the pixels into road and offroad, we search for the main road. This is a 2-D search, looking for road angle and offset. Once we find the main road, we search for the biggest branch road. This is also a 2-D problem, since the branch can leave the main road at any point along its length and at any angle. Once we find the first branch, other branches are constrained to leave the main road at the same point, so additional searches are each 1-D, looking at all possible angles. Each search looks in the classified image for a road-shaped region which has a total probability of greater than 0.5, based

on the classification probabilities of its pixels. The search for branches terminates when none of the candidates have high enough probability. A variation on this process confines the search to the top portion of the image, since SCARF processes sequences of images as the vehicle moves and intersections first appear in the upper part of the images. Once an intersection has been detected in one image, the search in subsequent images can be confined to a smaller range of locations and angles, depending on vehicle motion between frames.

SCARF was the primary vision system that used the Warp supercomputer on board the Navlab. Our experience with Warp was somewhat mixed. In the early days of Warp, the software and hardware environment had not matured, and it was difficult to really use the full power of the machine. Close cooperation between our group and the Warp group led to some important changes in hardware and in the programming environment, which in turn gave much better performance.

The early development of SCARF used our Flagstaff Hill bicycle path as the main test site. In the past year, we have run SCARF on several more test sites. In particular, at DARPA's request we tested various road following methods on sites that did not have paved roads. SCARF performed very well on gravel and dirt roads, as expected. The most difficult case involved a dirt road in a forest, which was mainly distinguishable in the video images by having less leaf cover than the surrounding forest floor. The philosophy of tracking the entire road, and using the entire image, made it possible for SCARF to track the road even though there were no clear border lines. We also ran SCARF on unlined suburban streets, with excellent results.

A brief overview of SCARF, and a description of the systems of which it is a part, are included in Chapter 2 of this report, "The New Generation System for the CMU Navlab". The Warp work is presented in more detail in Chapter 4. Further details on SCARF may be found in Crisman's thesis [3].

1.2.2 YARF

Our system for following structured roads is called YARF, for Yet Another Road Follower (an earlier version was called FERMI [5]). YARF addresses the problems of navigating on networks of city streets. This task requires the following capabilities:

- robust detection of road features (painted stripes, pavement/shoulder boundaries, etc.);
- detection of changes in feature appearance, such as changes in the color or continuity of a painted line;
- detection of changes in lane geometry, such as the addition of a turn lane near an intersection; and
- recognition of intersections, and path planning for navigation through them.

Current systems are limited in their ability to achieve these capabilities for several reasons. First, they typically use a single segmentation technique. Any single method will fail under certain circumstances, limiting system performance. Second, most systems fail to determine the confidence with which the road has been detected. Many current systems will follow the results of incorrect segmentations until the vehicle has driven off the road. Third, they have no model of the lane structure of the road. Driving on city streets requires that the system understand the semantics of the lanes. Finally, many systems have limited capabilities for intersection navigation. Intersections cover a large area compared to typical camera fields of view, and current systems process only one image at a time.

The YARF system addresses these problems in the following way:

- Multiple, specialized segmentation techniques for robustly extracting different kinds of road features. On typical rural roads, the double yellow line in the center of the road is detected by looking for pixels which have a yellow hue, while the white stripe on the right side of the lane is being tracked by searching for a bright bar of a specified width at a specified orientation.
- Explicit detection of segmentation failures and the analysis of possible causes of failures to detect changes in road appearance and the approach of intersections.
- An explicit model of the lane structure of the road. This model focuses processing on windows in the

image in which features should appear. It contains semantic information about the lane structure. It also provides the geometric model for the combination of the individual feature location measurements into a single estimate of the vehicle position on the road.
- Use of data from multiple images calibrated into a single vehicle-centered coordinate system for recognizing intersections and navigating through them.

Progress in the last year has included new trackers, models of road curvature, and road model fitting over multiple images. The combination of these new approaches and techniques has enabled us to increase our maximum demonstrated speed from 1 kph to 15 kph, with all processing on a single Sun4. Parallel versions, still being tested, hold the promise for further speed increases. YARF is described in Chapter 5, "YARF: A Progress Report".

1.2.3 The EDDIE Architectural Toolkit and Annotated Maps

Our new system, EDDIE (Efficient Decentralized Database and Interface Experiment), marks a turning point away from centralized, standardized architectures, to a flexible architectural toolkit. Our previous architectures concentrated on geometric reasoning and centralized, anonymous communications. EDDIE decentralizes those functions. Instead of all data flowing through a central map module, communications are now point to point. This allows the faster communications needed for reflex-level actions, while separating map-based reasoning into a dedicated module.

EDDIE is not a complete architecture, in the sense that it does not enforce a standard for how all robots ought to be built. Instead, it provides building blocks for communications, and for system start-up, monitoring, and control. In addition, EDDIE uses and supports the new mechanisms of annotated maps and of the integrated controller, described below.

EDDIE has been used to build several different architectures. The simplest systems use only a single perception module and the controller to do road following. These systems use the built-in position tracking of the low-level controller to monitor vehicle motion during image processing, and the smooth control modes of the controller to track commanded paths. More complex systems add modules, such as an "emergency stop" module that uses ERIM range data to find obstacles in the vehicle's path. The most complex systems we have built with EDDIE use several different road following modules, plus landmark detection, emergency stop, and map position update, along with Annotated Maps for mission planning and execution. A description of EDDIE is included in Chapter 2, "The New Generation System for the CMU Navlab".

Integrated Controller: Real-time mobile robot controllers have usually been designed with an emphasis on control theory ignoring the importance of system integration. Our new controller is based on the philosophy that useful mobile robots require a real-time controller with a wide range of capabilities in addition to control theory. These capabilities include:

- position estimation,
- mapping and tracking of paths,
- human interfaces,
- fast communication,
- multiple client support,
- and monitoring vehicle status for safety and debugging.

We have designed and implemented a controller framework that supports these capabilities. Using this framework, individual modules such as a position estimator, a path tracker, a mapper, network servers and other crucial elements have been successfully integrated into a controller for the Navlab autonomous vehicle. The controller incorporates an inertial navigation system into the low-level control loop, to provide accurate position estimation and path tracking. These capabilities are integral to EDDIE, freeing other modules from real-time tasks that properly belong to the lowest-level controller, close to the hardware. Amidi's master's thesis [1] discusses the results of trials with

different strategies for steering control, velocity control, and controller design.

Annotated Maps: EDDIE does not have a global map at the center, as does CODGER. Local positions, used only for the purposes of obstacle avoidance or path following, are never written into a map. Global, permanent, maps are handled by the separate mechanism of "annotated maps".

Besides the usual geometric data, annotated maps provide a mechanism for storing arbitrary bit fields, and associating those "annotations" with particular objects or locations. The information is then either retrieved on request, or automatically sent to a particular module by the "trigger" mechanism when the vehicle arrives at a specified location. In typical situations, annotations are used to describe the appearance of roads and landmarks. Triggers are used to indicate changes in vehicle control and sensor processing during the course of a mission.

The most ambitious mission we have performed to date is a 0.4 mile run on unmodified suburban streets in Pittsburgh's North Hills. This involved:

- Driving along curving suburban streets, with no pavement markings, including many different types of driveways;
- Traversing four intersections, at two of which the Navlab had to make a 90 degree left turn;
- Stopping for unexpected obstacles, and resuming motion when clear;
- Locating landmarks for position updates and for finding the destination.

We built a map of the route, driving the Navlab by hand and using the laser scanner to record the location of 3-D objects. Object positions were measured in multiple images, to discard moving objects (pedestrians, cars, dogs) and to improve the accuracy of measured position. The resulting map was annotated with triggers that controlled vehicle path execution. During the run, the vehicle started moving slowly, while it found landmarks to initialize its position. A trigger then caused the vehicle to speed up until it approached the first turn. At that point, triggers caused various modules to slow the Navlab, find 3-D objects, match them against the map, and update the vehicle's position estimate. Through the turn, vision was not able to see the road, so another trigger caused dead reckoning to take control until the vehicle was lined up with the next road, when the road was again in the field of view and vision could resume control. The run proceeded in this fashion until the final triggers, which matched the mailbox at the destination with the map, and brought the vehicle to a stop.

Detail on annotated maps may be found in Chapter 3, "Annotated Maps for Autonomous Land Vehicles".

1.2.4 Range Data Analysis

We have continued the development of a robust map building system for the Navlab. The maps produced by the system are stored in annotated maps. Our map building is made robust by

- Using the position information from the INS.
- Matching well-defined discrete objects between frames before attempting to match terrain descriptions.
- Representing explicitly uncertainty and confidence to produce an accurate map and to remove spurious items from the map.

Matching objects is not very expensive in our case because we have only a few objects to match in each frame and because we can assume that we have a reasonable estimate of the displacement between frames from INS or dead-reckoning so that the locations of the objects detected in one image can be easily predicted in the next image. The main issues are to remove spurious objects and to compute the location of the objects as accurately as possible. Spurious objects can be detected in two cases: Noise in the range image may cause the object detection program to hallucinate, and moving objects (e.g. people) crossing the field of view are detected as objects even though they should not be included in a map. Spurious objects must be eliminated because they may lead to disastrous results when they are used later on to correct the position of the vehicle. The position of the objects must be computed as

accurately as possible so that the position corrections that are computed using the object map are also accurate.

The problem of spurious objects is solved by calculating a confidence measure for each object. The confidence of an object is decreased if it is not found in an image in which it should appear based on previous observations and the current vehicle position, otherwise the confidence is increased. The objects with low confidence are discarded. Accurate object locations are obtained by updating the uncertainty on object location (mean and covariance matrix) each time an object is observed in a new image. The initial uncertainty is based on a sensor model and depends mostly on the distance between the object and the vehicle. The uncertainty also takes into account the fact that only a small part of the object surface is observed. The uncertainties are combined using standard maximum likelihood techniques.

The same techniques are used to identify specific objects in the map and to correct the vehicle position as it traverses a pre-stored map: the observations are matched with the information stored in the map. The matching uses several observations to allow for uncertainty computation and removal of spurious object detection through confidence evaluation. The map building and matching is now integrated into the annotated map navigation system and has been demonstrated in complex navigation scenarios.

1.3 Open Problems and Current Work

The results presented in this report cover the first year of an ongoing research program. SCARF is currently not active, and the EDDIE toolkit is currently stable. We continue our work in YARF, in annotated maps, and in range data analysis. In addition, we have begun new projects in integrating multi-sensor data and in understanding neural networks for road following.

YARF. The thesis work coming in YARF involves diagnosing failures of the trackers. If a white stripe is not detected as predicted, it could be because it is temporarily occluded, or is in deep shadow, or because the road markings changed, or because the road widens or turns abruptly. Some clues about the failure come from individual trackers: if the tracker window is all very dark, for instance, the vehicle may be entering a shadow. Other cues are global: if the detected location is drifting outward, perhaps the road is widening. Combining local reasoning, about appearances, and global reasoning, about geometry, will give YARF increased capabilities to understand the situation, update its models, and continue the run.

Annotated Maps. Our prototype implementation used a very low-level user interface, that allows unlimited flexibility in specifying each annotation, but requires almost unlimited typing and mouse pointing. We are first building a higher-level interface with macro capabilities, so we can define packages such as "turning at an intersection" that combine all the triggers typically used. In addition, we will build a computer interface, so autonomous or computer-aided mission planning systems will be able to generate annotations. Finally, we are expanding our annotated maps for off-road navigation. Instead of triggering a particular action when the vehicle crosses a line across the road, we will build triggers that fire when the vehicle enters or leaves a designated polygon in the terrain.

Range Data. We have used INS (inertial navigation) data for matching discrete objects, and we have separately developed the locus method for matching terrain patches and building maps. We will now combine the two ideas, using INS information to seed the locus search. The locus search finds the best transform between two map patches by an iterative process, which measures the residual match error and updates the transform. The search process is currently slow, since it must consider all six degrees of freedom (three translation and three rotation). We will be able to run much more quickly by constraining the search, especially with accurate angular information from the INS.

Integrated Positioning. Our map navigation experiments currently update vehicle position using only the most recent information. When 3-D landmarks are seen, their position relative to the vehicle is used as an absolute correction; when the road is visible, its perceived lateral position and orientation are assumed to be exact; and in other cases, the INS data is used. A better scheme would use data from each source, combined according to the error distributions for the individual position updates. We already have estimates of the precision of our INS system, and of the 3-D landmark location measurements. We will use a Kalman filter to keep a running track of the best estimate of vehicle position, and of the error margins in that estimate.

Understanding Neural Nets. Under separate funding, we have driven the Navlab using neural nets to track the road in video images. We are beginning a set of experiments to understand what features the neural net "hidden units" are matching, and whether we can achieve similar or superior performance by directly programming those feature detectors, rather than learning weights.

1.4 Theses

Omead Amidi, "Integrated Mobile Robot Control", Master's Thesis, Department of Electrical and Computer Engineering.

Jill Crisman, "Color Vision for the Detection of Unstructured Roads and Intersections", PhD Thesis, Department of Electrical and Computer Engineering.

InSo Kweon, "Modeling Rugged Terrain by Mobile Robots with Multiple Sensors", PhD Thesis, Robotics PhD Program.

Anthony Stentz, "The NAVLAB System for Mobile Robot Navigation", PhD Thesis, School of Computer Science.

1.5 Personnel

Supported by this contract or doing closely related research:

Faculty: Martial Hebert, Takeo Kanade, Chuck Thorpe

Staff: Mike Blackwell, Thad Druffel, Jim Frazier, Eric Hoffman, Ralph Hyre, Jim Moody, Bill Ross, Hans Thomas

Graduate students: Omead Amidi, Jill Crisman, Jennie Kay, Karl Kluge, InSo Kweon, Dean Pomerleau, Doug Reece, Tony Stentz

1.6 Publications

Selected publications by members of our research group, supported by or of direct interest to this contract.

[1] Omead Amidi.
Integrated Mobile Robot Control.
Technical Report, Robotics Institute, Carnegie Mellon University, 1990.

[2] Didier Aubert and Charles Thorpe.
Color Image Processing for Navigation: Two Road Trackers.
Technical Report CMU-RI-TR-90-09, Robotics Institute, Carnegie Mellon University, 1990.

[3] J. Crisman.
Color Vision for the Detection of Unstructured Roads and Intersections.
PhD thesis, Carnegie-Mellon University, 1990.

[4] Jill D. Crisman and Jon A. Webb.
The Warp Machine on Navlab.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 14.

[5] Karl Kluge and Charles E. Thorpe.
Explicit Models for Robot Road Following.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 3.

[6] Eric Krotkov, Reid Simmons, and Charles Thorpe.
Single Leg Walking with Integrated Perception, Planning, and Control.
In *IROS 90.* IEEE, July, 1990.

[7] InSo Kweon.
Modeling Rugged Terrain by Mobile Robots with Multiple Sensors.
PhD thesis, Carnegie-Mellon University, 1990.

[8] Dean A. Pomerleau.
Neural Network Based Autonomous Navigation.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 5.

[9] Dong Hun Shin and Sanjiv Singh.
Vehicle and Path Models for Autonomous Navigation.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 13.

[10] T. Stentz.
The NAVLAB System for Mobile Robot Navigation.
PhD thesis, Carnegie-Mellon University, 1989.

[11] Anthony Stentz.
The CODGER System for Mobile Robot Navigation.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 9.

[12] Anthony Stentz.
Multi-Resolution Constraint Modeling for Mobile Robot Planning.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 11.

[13] Hans Thomas, David Wettergreen, Charles Thorpe, and Regis Hoffman.
Simulation of the Ambler Environment.
In *23rd Pittsburgh Conference on Modeling and Simulation.* IEEE, May, 1990.

[14] A. Stentz and C. Thorpe.
Against Complex Architectures.
In *Proc. 6th International Symposium on Unmanned Untethered Submersibles.* June, 1989.

[15] Charles E. Thorpe.
Outdoor Visual Navigation for Autonomous Robots.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 15.

[16] Charles E. Thorpe.
Vision and Navigation: the The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990.

[17] David Wettergreen, Hans Thomas, and Charles Thorpe.
Planning Strategies for the Ambler Walking Robot.
In *IEEE International Conference on Systems Engineering*. IEEE, August, 1990.

Chapter 2: Toward Autonomous Driving: The CMU Navlab

Charles Thorpe
Martial Hebert
Takeo Kanade
Steven Shafer

2.1 Introduction

The goal of the CMU Navlab project is to build autonomous systems capable of outdoor navigation, both on roads and cross-country. Since the outset of the project in 1984, we have held two main tenets: the importance of complete systems, and the importance of focusing on bottlenecks. Our emphasis on complete systems has meant that, since the beginning, we have closed the loop from sensing to action, in realistic outdoor scenarios. We have been forced to deal with the vagaries of natural illumination, of bright sunlight and clouds and rain and snow; and we have had to confront the problems of camera calibration, path planning, real-time computing power, and software system architectures. While the logistical costs of performing such real experiments have sometimes been significant, our resulting algorithms and systems are calibrated to reality. Our second principle, of focusing on the bottlenecks, has pushed us to work on the most difficult problems first. For outdoor navigation, the biggest challenge, and our main area of research, has been in image understanding in difficult conditions. Instead of running at high speeds on cleanly-marked expressways, we have worked on unstructured roads (including dirt roads and a winding asphalt bicycle path), on the changing appearance of structured roads in dappled shadows and at intersections, and on off-road navigation over rough terrain. Once we had the first versions of reliable perception software, we also developed novel planning methods for rough terrain, and have designed and built systems software to forge the separate perception and planning modules into integrated systems. Other technologies, such as vehicle design, high-speed computing, and control theory, are not the main bottlenecks. While important components, they have been or are being developed by other groups, often outside the mobile robotics community. By directly confronting the central areas of perception, planning, and system-building for mobile robots, we are completing the missing links that will enable us to build the reliable high-speed mobile robots of the future.

We now have significant results in many of those areas. Our Navlab robot van (shown in Figure 2.1) drives itself at slow speeds along unmarked, unmapped trails, locating and traversing intersections. On more typical structured roads, the Navlab drives up to its mechanical limit of 28 kph. It can run without a map, or use maps it has built, along with information from previous runs, to select different behaviors at different locations. Off road, the Navlab can move slowly over moderately rough terrain, and can map large areas as it drives. The resulting software has been transferred to other projects, including the DARPA Martin Marietta ALV and our own NASA-sponsored AMBLER, a walking machine for planetary exploration.

2.1.1 Context

Our work is part of the broader framework of DARPA's Strategic Computing Initiative, including the Autonomous Land Vehicle (ALV) project that began in 1984. Several of the contractors from Strategic Computing Vision worked on perception and planning for autonomous navigation. Among others, the University of Massachusetts and Honeywell developed motion tracking software [3, 6]; SRI developed tracking using 3-D data [7]; ADS built Qualitative Navigation [29]. Martin Marietta built and operated the ALV vehicle itself, and developed their own road following software [41]. Hughes and the University of Maryland contributed off-road and on-road navigation, respectively, directly to the ALV testbed [20, 42]. The role of CMU was to build a New Generation Vision System. We were tasked to look beyond the immediate problems of getting the ALV through its first demonstrations, and to address the issues of more difficult perception and integration.

Figure 2.1: The Navlab

Beyond the DARPA community, the past five years have seen several other outdoor mobile robot projects. In the US, Texas A&M has begun work on visual tracking for convoy following and obstacle avoidance [19]. General Motors is working on lane following at high speeds, under relatively constant illumination [21, 22]. In Germany, Professors Dickmanns and Graefe [32] have built an elegant control formulation for driving on autobahns. Fujitsu and Nissan in Japan have built prototype road-following software [33]. The research at CMU, and within the DARPA community, is distinguished from all of these by its concentration on the more difficult vision problems of bad weather, bad lighting, and bad or changing roads.

2.1.2 Navlab Testbed Vehicle

The Navlab vehicle was designed and built in 1986 to provide a testbed for vision and navigation experiments. It is based on a standard commercial van, with a rooftop air conditioner, plus one or more video cameras and a laser rangefinder mounted over the cab. On the inside, it is a computer room, with five electronics racks, 20kw of onboard power, and miscellaneous consoles and monitors. Over time, the Navlab has carried Sun 3's, Sun 4's, several generations of the CMU / GE Warp supercomputer, various specialized real-time controllers, gyrocompasses, an inertial navigation system, and a satellite positioning system. We currently use only a small portion of the available rack space and electrical power. Our current real-time controller occupies four slots in a VME cage, and our general-purpose computing consists of three Sun 3's in a single cage, and two Sun 4's in another cage.

The most important payload of the Navlab is the researchers. There is always a safety driver in the driver's seat, watching over the Navlab's autonomous runs. In the back, there is room for five researchers plus observers. The quality of our mobile robot software increased greatly when the graduate students and engineers were able to ride along on autonomous runs, partly out of self-preservation, but mainly because they could see and feel how their code worked. We run standard SunOs Unix[1] on the Navlab, so we have a standard programming environment and tools to find and fix bugs in our programs during an experimental run (debuggers, editors, compilers, etc.).

2.2 Color Vision for Road Following

Roads that are nearly straight, evenly-illuminated, and well-marked, can be tracked easily in color or monochrome video images. Finding the edges of a clean sidewalk, or tracking freshly-painted white stripes on an empty expressway, are both straightforward. Road following becomes much more difficult when the road runs through dappled shadows, or when illumination suddenly changes as the sun goes behind a cloud, or when the "road" is a meandering bicycle path with no lines or stripes and with broken and uneven borders. Therefore, the challenge in building truly autonomous road following systems is to be able to handle a variety of road conditions and changing illumination. To illustrate the problem, our first road following software ran a simple edge detector (Roberts' operator, followed by thresholding) over the image, and looked for edge fragments that had strong contrast, were parallel, and pointed in roughly the correct direction. This worked very well for clearly-marked sidewalks. When we took our robot onto a bicycle path, the highest-contrast edges in the scene were shadow edges. At the right time of day, the shadows of tree trunks fell along the road, producing strong, straight, parallel edges at nearly the predicted direction. Our road-following software turned into tree-shadow-following software.

Navlab test sites include a variety of road conditions, from dirt roads to freeways. A single perception system would not be able to address all possible configurations. Instead, our approach is to build different systems for unstructured roads, such as dirt roads, and structured roads, such as highways and city streets. This allows us to take advantage of the road structures when they are available while retaining the ability to deal with unstructured roads when needed.

A first system, SCARF, deals with unstructured roads. The SCARF system uses adaptive color classification. It deals with changing illumination and changing road appearance by updating its color models for each new image. It handles poorly-defined roads by classifying all the pixels in the image, and by using a simple road model in a voting scheme to find the most probable road in the image.

YARF, our second vision system, deals with structured roads, such as highways and city streets. It takes advantage of the lines and stripes of structured roads, and uses an explicit model of those features both to guide individual trackers and to filter and validate its detected road model.

SCARF and YARF do not require any external input expect to bootstrap the system at the beginning of a run. A road following system that can be trained on a section of road prior to a mission should be faster and more reliable. To investigate this idea, we have built ALVINN, the third main color vision system currently running on the Navlab, which uses a connectionist architecture. It achieves its power by being trained directly on the current road, and by processing quickly so that small imperfections tend to be smoothed out.

[1]SunOs is a trademark of Sun Microsystems, and Unix is a trademark of Bell Labs

2.2.1 SCARF

Three approaches have been used in unstructured road following: edge extraction, thresholding, and classification. In systems that use edge extraction, gradient operators are applied to the image of the road. Strong edges are assumed to correspond to road edges and are grouped to yield road geometry [42]. Edge-based systems can be very fast and can work well on clearly delineated roads with no shadows. As soon as strong shadows appear, however, they break down rapidly because strong edges now correspond to shadow edges. Systems that use thresholding use some combination of the color bands, e.g., red - blue, and threshold the resulting image [17, 26, 41]. Those systems are also limited by shadows. They label all pixels with similar intensities as road. But when shadows are present, shaded road and shaded off-road often have very similar features, thus confusing the classification.

Our approach to avoid shortfalls of those previous systems is to use adaptive color classification. We have built a system, SCARF, which stands for Supervised Classification Applied to Road Following, to demonstrate the performance of this approach [10]. SCARF runs in a loop of: classify image pixels, find the road model that best matches the classified data, and update the color models for classification. The simple models of road color and geometry make very few assumptions about the road, and allow SCARF to run robustly even when following unstructured roads.

The first strength of SCARF comes from representing multiple color classes, as Gaussian distributions in full RGB color, and from calculating probabilities instead of using binary thresholds. SCARF typically uses four color classes to describe road appearance, and four for off-road objects. In the classification step, each pixel is compared to all eight classes. The output of classification is both the label of the most probable class, and its probability. Having multiple classes allows SCARF to represent the different colors of the road (for instance asphalt, wet patches, shadowed pavement, and leaves) and off-road objects (trees, sunlit grass, shaded grass, and leaves). Using full color, instead of monochrome images or some combination of colors, keeps all the image information that may be useful in discrimination. The Gaussian representation of each color class says, intuitively, whether a particular variation in color is significant. Sunlit asphalt tends to be homogeneously colored, and is represented by a class with small variance; grass has more variety, and is represented with correspondingly larger variances. Having Gaussian representations of the colors for each class makes it possible to calculate the likelihood that a given pixel belongs to a particular class. While most other navigation systems simply use binary thresholds, SCARF gives the probability for each classified point. This is especially important for cases such as dry leaves that occur both on and off road, for example. A particular pixel may somewhat more closely resemble offroad leaves than onroad leaves, but the confidence that it should be classified as offroad will be very close to the confidence that it should be classified as onroad, so that the pixel will (correctly) contribute very little to the overall road location determination.

Classified pixels vote for all road locations that would contain them, with votes weighted by classification confidence. The road with the most votes is used both for steering, and for recalculating the color classes using nearest mean clustering to collect new road and offroad color statistics. SCARF uses a simple model of road geometry. Roads are represented as triangles in the image. The apex is constrained to lie on a particular image row, corresponding to the horizon, and the base of the triangle has a fixed width, dependent on road width and camera calibration. There are two free parameters: the column in which the apex appears, and the skew of the triangle in the image. While this simple 2-parameter model does not represent curves or hills or road width variations, it does approximate the road shape well enough to allow reliable driving. It is especially effective because the voting procedure uses all pixels, not just those on the edges, and is therefore relatively insensitive to misclassifications. A model with more free parameters could represent more potential road shapes, but would often be led astray and find curves or branches where in fact all that exists is noise. Furthermore, the simple model allows for fast voting and functions well with small amounts of data, so SCARF can process highly reduced images (typically 60 by 64 or 30 by 32) at high rates (approximately two seconds per frame). Processing images closely spaced along the road means that small errors in road representations are corrected before the vehicle arrives at the mistaken locations.

Processing images closely spaced in time means that even the drastic illumination changes caused by clouds covering the sun appear as gradual shifts in road appearance, and so do not derail SCARF.

The basic SCARF system runs on Sun workstations. It was demonstrated on a number of different roads. SCARF has driven the Navlab along bicycle paths, dirt roads, gravel roads, and suburban streets. SCARF has been integrated into several of our Navlab systems. Figure 2.2 shows SCARF correctly finding the road even through deep shadows, where the road is not obvious even to a human observer. It successfully demonstrated that pixel classification based on Gaussian representations of color classes is appropriate for road navigation in the presence of strong shadows and changing illumination.

We have built several extensions of SCARF. The first extension is to use parallel hardware instead of conventional workstations to improve preformance. We have implemented SCARF on the WARP computer, a ten-cell systolic array. SCARF is parallelized by dividing the image into strips, and by processing each strip on a separate cell. The second extension to SCARF is to add to the road model by checking for intersections as well as for the main road. Figure 2.3 shows SCARF finding an intersection in a series of images as the vehicle approaches the branch point.

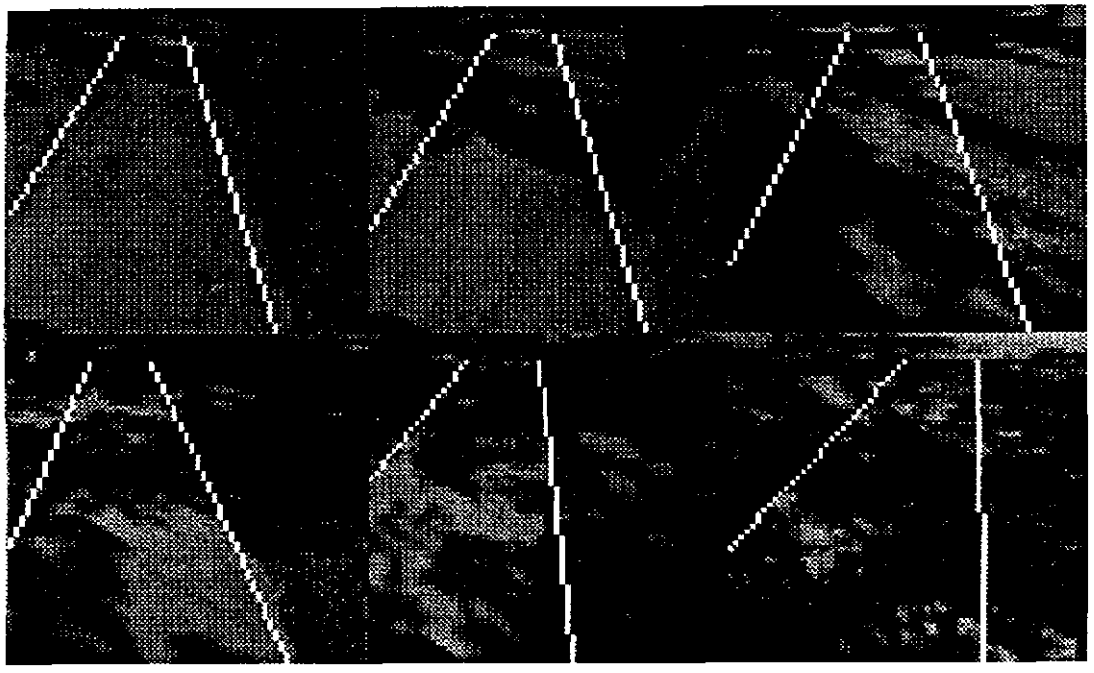


Figure 2.2: SCARF correctly finding the road in difficult shadows

2.2.2 YARF

The problem of road following in urban environments requires specific techniques to take advantage of the prior knowledge of the environment, e.g. well-marked, highly structured highways. Several systems for following structured roads have been developed. The VaMoRs system [13]combines specialized hardware with a control formulation of the problem to achieve runs of up to 96km/h. VaMoRs uses simple feature trackers to track the position of road center line and side markings. However, it may be sensitive to changes in illumination, shadows, and changes in road structure such as intersections. The LANELOK system [21, 22] can use three different types of image operators to track road edges, Sobel edge detection followed by Hough Transform, region extraction, and template matching. LANELOK has been demonstrated off-line on thousands of images. In the system from the University of Bristol [36], lane markings are extracted as regions of the image that are brighter than a given

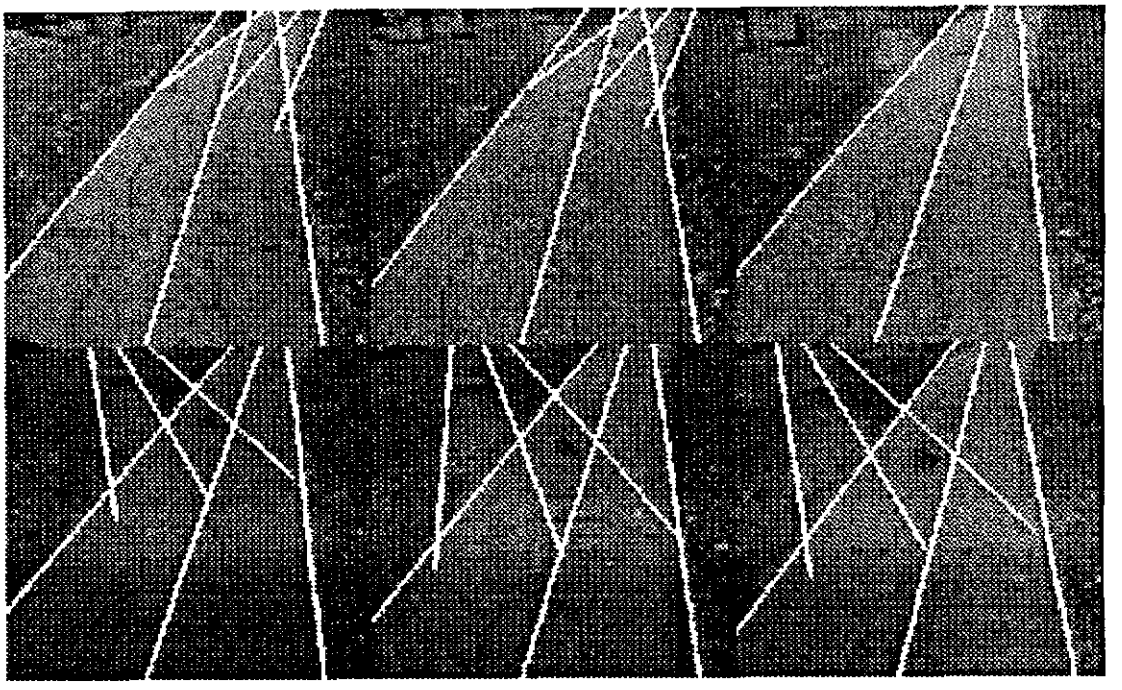


Figure 2.3: SCARF finding an intersection

threshold and limited by edges of appropriate geometry. A circular arc is fit to the regions after backprojection on the ground plane.

All these systems are limited by the use of a single segmentation technique to locate the road. Therefore, they do not have any mechanism for recovery in the event that this particular segmentation technique should fail. To address this problem, we have developed the YARF system (Yet Another Road Follower) which explicitly models as many aspects of road following as possible, for driving on structured roads [25, 4]. Highways, freeways, rural roads, even suburban streets have strong constraints and easily identifiable features. For instance, the road center line is yellow, has a constant known width, and its curvature is lower than a known threshold. The key idea of YARF is to model explicitly each individual constraint and feature. From feature models we build specialized image operators that find features such as road markings. From constraint models, we build specialized trackers that apply the image operators on long sequence of images by predicting the location of a feature in an image from its location in the previous image. A tracker for the road center line finds a yellow stripe in a small window in a color image and predict its location in the next image using a model of road geometry and current vehicle motion. Also, constraint models provides a way to detect and recover from errors in feature detection. This makes reasoning easier and more reliable. When a line tracker fails, for instance, an explicit model of road and shoulder colors adjacent to the line helps in deciding whether the line disappeared, became occluded, turned at an intersection, or entered a shadow. This kind of geometric and photometric reasoning is vital for building reliable and general road trackers. In addition to geometric constraints, YARF uses an explicit model of the noise of feature detectors and vehicle position to yield optimal performance.

YARF has four main components: feature trackers, geometric modeling, error detection and recovery, and noise modeling.

Specialized Feature Trackers

YARF has individual operators that know how to model and track specific features, such as road edge markings (white stripes); road center lines (yellow stripes); and shoulders. YARF also uses an explicit geometry model of the road, consisting of location of vehicle on road; location of stripes; type of stripes (e.g. broken or solid); and

maximum and current road curvature. Other features, which are not yet modeled but which may be helpful, include locations of shadows, 3-D effects as the road goes through valleys and over hills, and global illumination changes.

The yellow line tracker, for instance, uses the hue of the lines for segmentation. The hue is calculated for all pixels within a window around the predicted line location. Pixels with a hue between 40 (reddish-yellow) and 100 (greenish-yellow) are set to 1, others to 0. The results of this thresholding are quite noisy. Pixels that are very close to gray have an unstable hue value, while yellow lines in dark shadows are often so dark that digitization noise nearly swamps their yellow hue. As a result, the images after hue thresholding often have isolated noise points, both false positives and false negatives. We clean up the output with a "shrink and grow" operator. The resulting image is normally dominated by one or two blobs, corresponding to the yellow line or lines. The blob descriptors are returned as the line locations. Figure 2.4 shows the yellow line tracker, and a separate white line tracker, finding the road lines even in complex shadows.

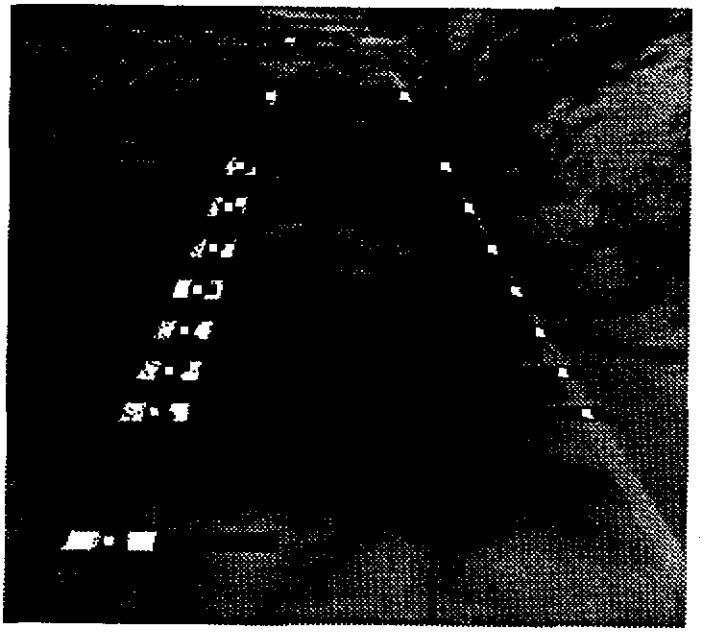


Figure 2.4: YARF tracking yellow and white lines in complex shadows

Road Geometry

YARF is designed for higher speeds than SCARF, and runs in a more predictable environment. This requires and allows a more complex road model, that encodes curvature as well as position. YARF models the road as a generalized stripe, that is a one-dimensional feature that is swept perpendicular to a spine curve. The spine is modeled locally by a circular arc assuming that the road lies on a flat ground. The equation of the spine is found by fitting a circular arc to the detected features. Since, the equation of a circle

$$radius^2 = (x - xcenter)^2 + (y - ycenter)^2$$

is nonlinear, and sensitive to noise, we can approximate a circular arc with a parabola, similar to the approach of Dickmanns [32]:

$$x = half\_curvature \times y^2 + slope \times y + lateral\_offset$$

The least-squares values of curvature, slope, and lateral offset are easily computed using the matrix pseudo-inverse. Since this is done in vehicle coordinates, with the vehicle pointing approximately along the road, the parameters calculated are good approximations to those of the best-fit circle. In practice, this approximation is adequate for the sorts of curvatures and slopes of roads within the Navlab's field of view. To improve the stability of the estimation, we fit the curvature model to features detected over a few frames.

Error Detection and Recovery

Occasionally features are found incorrectly. YARF detects these mistakes both locally, based on the results of a single operator, and globally, checking for consistency. Local error detection depends on the specific operator. Some operators, such as the oriented window tracker, can only report a correlation measure as a confidence. Others, such as the two-color blob detector, can provide a little more information. The blob detector usually finds a light blob (the white line) against a dark background (asphalt). It maintains statistics of the mean, variance, and covariances of red, green, and blue, for both feature and background colors. If all pixels in its prediction window are the background color, the color blob detector reports a missing feature. If a light-colored blob is found, but only at the edge of the window, it reports a clipped feature. If all pixels are much lighter or darker than modeled by either color, it reports an illumination shift. It is up to the higher-level calling program to decide whether the road has widened, the white stripe is temporarily missing, or the lighting really has changed.

Global error detection uses the output from all feature detectors is a single frame. There are many ways to check for data consistency. The simplest, performing a least-squares fit and examining the residual, gives some cues as to whether there is an outlier, but does not reliably indicate which point is in error. Better approaches come from the "robust statistics" literature.

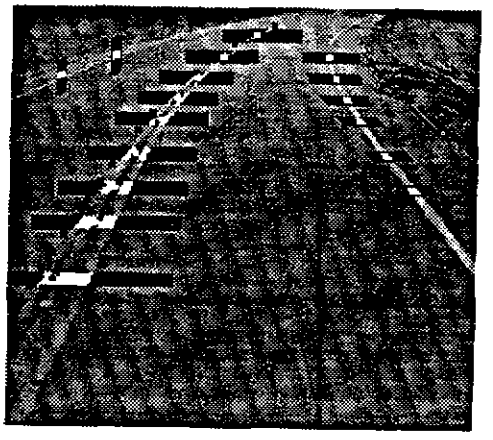
Error detection and recovery in a critical component of YARF. It allows for robust navigation in the presence of changing illumination, shadows, and noisy road features while using fast and simple specialized trackers.

Noise Modeling

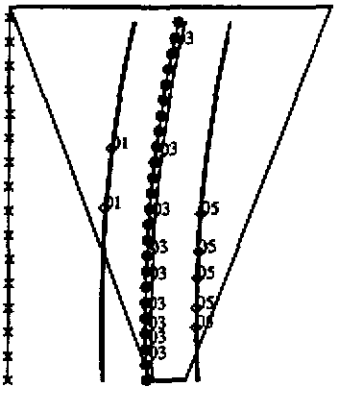
We have achieved good results by fitting curves to the points detected over the three most current frames, with no error weighting or filtering at all. For instance, Figure 2.5(a) shows the features (center line and edge markings) extracted from an road scene, Figure 2.5(b) shows the road the road model fit to the features using straight least-squares. In Figure 2.5, the diamonds show tracked left and right lane lines, and the solid circles show estimated road position and heading. For the relatively slow speeds (up to approximately 15 kph) of YARF, and with the Navlab's accurate dead reckoning, the errors in detected positions are probably dominated by image processing noise rather than vehicle motion noise. Future runs, at higher speeds, will probably require more elaborate filtering schemes. We are working on two approaches to noise filtering: Kalman filters, and robust statistics.

The intuition behind a Kalman filter is that the current estimate of state is a combination of current measurements and the previous state estimate, transformed to account for system dynamics. The weight given to current measurements depends on their believed accuracy. The weight given to the previous estimate depends both on how accurate the previous estimate was thought to be, and on how accurately it can be transformed into current coordinates. In the case of road following, the weight for current detected features comes from the accuracy of feature detection and camera calibration. Vehicle motion errors can be reduced by inertial sensing. Road model errors will depend on the situation. For the gentle curves and smoothly varying curvature of an interstate, prior estimates can be extrapolated for long distances, and can carry the vehicle through shadows and other visually confusing areas where current tracking fails. For the winding turns of country roads, however, the road model can change radically over very short distances, and the weight given to prior models in the filter equations must decrease rapidly.

Kalman filtering reduces the influence of noise in the system but it does not remove outliers, i.e., measurements that far from the real value. Robust statistics provide a way to eliminate outliers by minimizing a function of the least-squares residuals that falls off more rapidly for large residuals. Outliers are eliminated because they do not contribute to the overall sum of residuals. We are currently evaluating such a technique, M-estimation, in YARF.



(a) Extracted center and edge features



(b) Road model derived from features extracted above. Diamonds show detected feature locations, solid circles show derived center line.

Figure 2.5: YARF tracking result

Performance

YARF has driven the NAVLAB at speeds of up to 15.0 mph on a public road. The environment used in the experiments contained large shadows generated by surrounding trees. It also including relatively high-curvature curves, thus demonstrating the preformance of the road model.

YARF works because of the integration of all its modeled constraints. Explicitly modeling tracker performance and feature appearance, and using specialized trackers, allows high speeds, high accuracy, and local failure detection. Explicitly modeling road geometry allows accurate predictions, and enables global error detection. Explicitly modeling errors and uncertainty allows YARF to correctly size its prediction windows. And explicitly modeling changes in road geometry gives YARF the capability to handle urban streets, with intersections and other discontinuities in lane structure and appearance, as well as the cleaner environment of highway lane following.

2.2.3 ALVINN

SCARF and YARF do not require any input other than video images except for bootstrapping the system. If we allow the system to be manually trained on a portion of the road to be followed, we should be able to build a faster and more robust system since more prior information on road appearance and geometry is available. ALVINN, for Autonomous Land Vehicle in a Neural Net, implements this idea. ALVINN was built by the CMU Connectionist group [35]. The weights in the hidden units of ALVINN's neural network are trained by driving the Navlab by hand. During the training phase, ALVINN inputs the camera image, and the steering angle from the human driver, at each time step. The image is pre-processed to enhance road contrast. The enhanced image and the steering angle

are fed to a back-propagation algorithm that adjusts weights in the hidden units, until the weights settle to values that give the correct steering response for each input image. Typically, training takes less than a hundred input images and uses less than 5 minutes.

With this training scheme, ALVINN directly learns how to follow roads. It is more difficult to train the network to recover from errors, when it is not quite aligned on the road. To provide examples of images from slightly different vantage points, and the proper steering commands, each input image is reused in several positions. The images are shifted to simulate a variety of errors, and the steering command is shifted to generate the command that would bring the Navlab back on to the road.

When ALVINN runs, it preprocesses the input images, and gives them to the net. ALVINN then directly outputs the steering wheel angles as dictated by the network, with no reasoning about road location. ALVINN uses reduced-resolution images (typically 30 by 32 or 45 by 48 pixels), and runs in about a fifteenth of a second per image.

One characterization of ALVINN is that it uses a compiled representation, going straight from images to steering with no intermediate geometric or symbolic representation. During its learning phase, the back-propagation algorithm automatically compiles this knowledge, by selecting the features that discriminate between different steering angles, which correspond to different road locations. Since ALVINN starts with no pre-conceived idea of what the road looks like, it learns different sets of weights to follow many different types of roads with no change in the underlying algorithms.

The disadvantage of a compiled representation such as that used by ALVINN is that it cannot take advantage of geometric or symbolic input. If ALVINN is trained to run on a particular road, it is impossible to tell it that a second road is just like the first, only twice as wide. Since there is no explicit representation of "road width", or even of "road", there are no symbolic parameters to be changed or manipulated. The advantage of such a representation is that it is fast, and is easy to train for a particular road. The weights learned by ALVINN tend to be large, low-frequency edge masks, or matched filters that look for the road in general locations. Thus, local imperfections in the road or in lighting do not greatly distort the output steering direction. Figure 2.6 shows the weights for one of ALVINN's hidden units. The square shows the weights coming in to one hidden unit from the input image, and the line at the top shows the weights going out to different steering angles. White indicates positive weights, and dark negative. This unit mainly looks for a road on the left edge of the image, and mainly votes for turning left. There is also a secondary pattern that would match a road further to the right, and slightly positive weights supporting straight ahead steering.

ALVINN is the fastest of our current road following systems, because of its compiled representation. It has also been the most difficult to integrate into systems, because it does not output detected road locations. While SCARF and YARF report the location and orientation of the road, ALVINN only produces steering commands. It is, however, possible to reason geometrically about ALVINN's steering output to infer some information about road location. The output arc will bring the Navlab onto the center of the road at some distance along the arc. This distance would in general be unknown, depending on the particular driving style used during training, except that the artificially shifted images, used in training the Navlab to return to the road, use a particular specified geometry. This geometry, used for shifting the roads to provide training examples, can also be used to measure how far along the steering arc the center of the road lies. The intersection of the arc and the road center gives a single point known to lie on the road center line. Observing several of these points over time allows us to approximate the apparent road position and shape, which allows the map-based reasoning needed to integrate ALVINN with other systems.

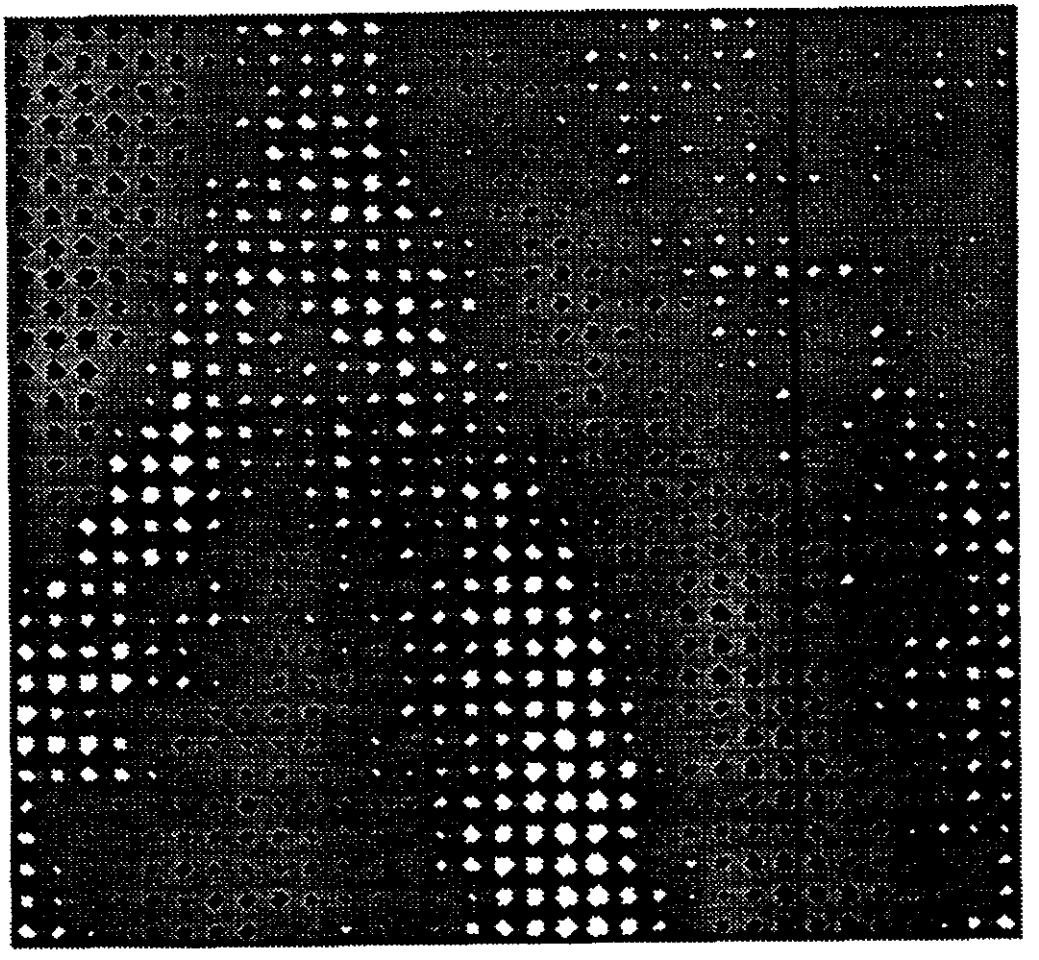


Figure 2.6: ALVINN weights for one hidden unit. Bottom: input weights. Top: output weights. Positive weights are white, negative are black.

2.3 3-D Perception

An outdoor mobile robot needs information derived from appearance (*e.g.* road location in a color image, or terrain type), but it also needs to know the geometry of the observed environment. In some tasks, such as cross-country navigation, the most important information is the geometry of the *terrain*, the set of 3-D surfaces observed or traversed by the vehicle. The first step towards building geometrical representations of a terrain is to choose a suitable sensor. Clearly a single color camera is not suitable for collecting 3-D data. An alternative is to use passive techniques for recovering 3-D data such as stereo vision. There are significant drawbacks to those techniques, including high computational demand, difficulty in ranging bland surfaces, and reliance on ambient lighting. Instead, we use an active sensor, a laser range scanner, which can generate a high resolution depth image of the terrain in front of the vehicle. Using such a sensor alleviates the need for inferring 3-D information from 2-D information and, being active, is also less sensitive to outside illumination. The technology used in the sensor is very recent and is not used very widely yet. We discuss the drawbacks, advantages, and prospects for future improvements in the technology in Section 2.3.1.

The terrain can be described at different levels of resolution depending on the task, the environment, and the amount of computation time allocated for 3-D perception in the system. For example, a system that follows roads that are known to be locally flat and mostly obstacle free, requires a completely different representation than a system that navigates through rugged open terrain. In the latter case, vehicle safety becomes the overwhelming issue while vehicle speed becomes much less important. This is consistent with the general approach that the components of a mobile robot must be tailored to environment and task.

In addition to several types of terrain representations, we also distinguish between techniques that involve building a representation from a single image, and techniques that put together representations from several images to build a consistent *map* of the terrain traversed by the vehicle. An example of the former is fast obstacle detection in which the goal is to detect unexpected objects in the current image as fast as possible. An example of the latter is building a 3-D map of a long stretch of terrain so that the system can later use the map to retraverse that area. Depending on the environment and the task, we should be able to build maps at all levels of resolution.

We have identified three types of representations that we discuss in detail in the following sections. For each type of representation we describe two types of processing: range data processing for building a terrain representation from a single range image, and matching techniques for building consistent maps from several observations.

1. *Discrete objects and obstacle detection:* A coarse description of the environment is one in which only discrete objects are represented without an explicit representation of the surrounding terrain. This representation is appropriate when navigating in mild terrain with clearly defined objects. There are two applications of this level of representation: fast detection of obstacles during road following, and building maps of the objects observed as the vehicle travels on a network of roads. Such a map may be used in a map-based navigation system to correct the vehicle position by matching observed objects with predicted objects from the map.
2. *Feature-based terrain modeling:* A finer description of the environment involves describing the terrain by a set of features (regions and edges) in addition to discrete objects. This representation is used for cross-country navigation, in which the vehicle, in order to navigate safely, must take into account the local shape of the terrain as well as discrete objects. The terrain features can be used in conjunction with discrete objects to build more reliable terrain maps over many images. This is particularly true in mild open terrain in which not enough discrete objects are observed to reliably match observations.
3. *High resolution terrain models:* The highest-resolution representation is achieved by building elevation maps, the spatial resolution of which may be as fine as 10cm. Large terrain maps can be built from individual high resolution maps by correlation-type matching. The advantage over feature- or object-based representations is that information about the local shape of the terrain is preserved everywhere. The price to pay is much higher computation time. Therefore, high resolution terrain maps are most useful in applications in which the computations can be done off-line, or applications in which stop-and-go motion of the vehicle is acceptable. The latter situation occurs when navigating through very difficult terrain in which case the best terrain description is necessary to ensure the safety of the vehicle.

2.3.1 Range sensing

The sensor that we use on the Navlab is the ERIM scanner (Table2.1). This scanner is typical of the class of laser range finders based on the measurement of the phase shift of an amplitude modulated laser [5]. Table 2.1 lists characteristics of both the ERIM, one of the earliest scanners, and the Perceptron, a later model.

Using an active laser range finder has considerable advantages over more traditional techniques such as stereo vision. It is insensitive to outside illumination, it is fast compared to the computation time required by standard passive techniques, and it provides a high resolution range map as opposed to the sparse map produced by most passive techniques. However, we have found a number of problems with this laser ranging technology that should be addressed in order for it to be widely used:

- *Mixed points:* At the boundary between two objects, one part of the laser spot, or footprint, is on one

	ERIM	Perceptron
Eye Safe	yes (?)	yes
Field of View	80h by 30v	60 by 60 (programmable tilt)
Pixels	256 by 64	256 by 256
Ambiguity interval	20 m	40 m
Depth	8 bits (8 cm)	12 bits (1 cm)
Intensity	8 bits	8 bits
Max range	40 m (?)	50 m
Scan rate	2 frames / sec	2 frames / sec
Scan direction	top to bottom	programmable
Interface	VME to Sun	VME to Sun
Temperature	narrow range	'Pittsburgh'
Construction	wire wrap	printed circuit
Components	all custom	most off the shelf
Size	90w by 35h by 45d cm	45w by 35h by 35d
Weight	50kg	< 25kg
Power	26VDC	110VAC

Table 2.1: Relative performance of example range scanners

surface while the other is on the other surface. Since the sensor integrates over the entire footprint, the resulting measured point does not lie on either surface but is "in between" the two surfaces. Such a point is a mixed point, because it is measured from a mixture of reflections from both surfaces. Mixed points are inevitable with the current technology. Most mixed points can be removed through median filtering. However, the mixed point problem implies that edges in range images, especially edges between distant objects, are highly unreliable. This should be taken into account in the choice of range image processing algorithms.

- *Acquisition rate:* The typical acquisition rate of two images/second is too slow. The motion of the vehicle can be significant while the image is being scanned, thus leading to a distorted range image. This is not a problem at very low speed, such as in cross country navigation, but may preclude the use of this type of sensing at higher speeds, such as in highway driving, and for tracking moving obstacles.
- *Sensitivity to surface material:* In early scanners, the measured range varied with surface type. More recent scanners can adjust for different uniform surfaces, but still have problems with edges between surfaces that are at the same depth but have different reflectances. The change in reflectance causes changes in internal sensor gains, which upsets the phase detection, which produces a spurious depth edge.

2.3.2 Discrete objects and obstacle detection

The lowest resolution terrain representation is an object map which contains a small number of objects represented by their trace on the ground plane. Several techniques have been proposed for obstacle detection. The Martin-Marietta ALV [14, 15, 41] detects obstacles by computing the difference between the observed range image and pre-computed images of ideal ground planes at several different slope angles. Points that are far from the ideal ground planes are grouped into regions that are reported as obstacles to a path planner. A very fast implementation of this technique is possible since it requires only image differences and region grouping. It makes, however, strong assumptions on the shape of the terrain. Specifically, it restricts terrain shape to a few admissible slopes and elevations. It also takes into account only the absolute positions of the potential obstacle points, not relative positions and slopes. As a result a short, sharp ridge or step would be overlooked, even though it may be an obstacle. Another approach proposed by the Hughes AI group [11] is to detect the obstacles by thresholding the normalized range gradient, $\Delta D/D$, and by thresholding the radial slope, $D\Delta\phi/\Delta D$. The first test detects the discontinuities in range, while the second test detects the portion of the terrain with high slope. This approach has the advantage of taking a vehicle model into account when deciding whether a point is part of an obstacle.

We use an elevation map approach to detect obstacles for the Navlab. Each cell of the terrain contains the set of

data points that fall within its field. We can then estimate the surface normal at each elevation map cell by fitting a reference surface to the corresponding set of data points. Cells that have a surface normal far from the vehicle's idea of the vertical direction are reported as part of the projection of an obstacle (Figure2.7). Obstacle cells are then grouped into regions corresponding to individual obstacles. The final product of the obstacle detection algorithm is a set of 2-D polygonal approximations of the boundaries of the detected obstacles that is sent to an A*-type path planner. In addition, we can roughly classify the obstacles into holes or bumps according to the shape of the surfaces inside the polygons.

Figure 2.8 shows the result of applying the obstacle detection algorithm to a sequence of ERIM images. The Figure shows the original range images (top), the range pixels projected in the elevation map (left), and the resulting polygonal obstacle map (right). The large enclosing polygon in the obstacle map is the limit of the visible portion of the world.

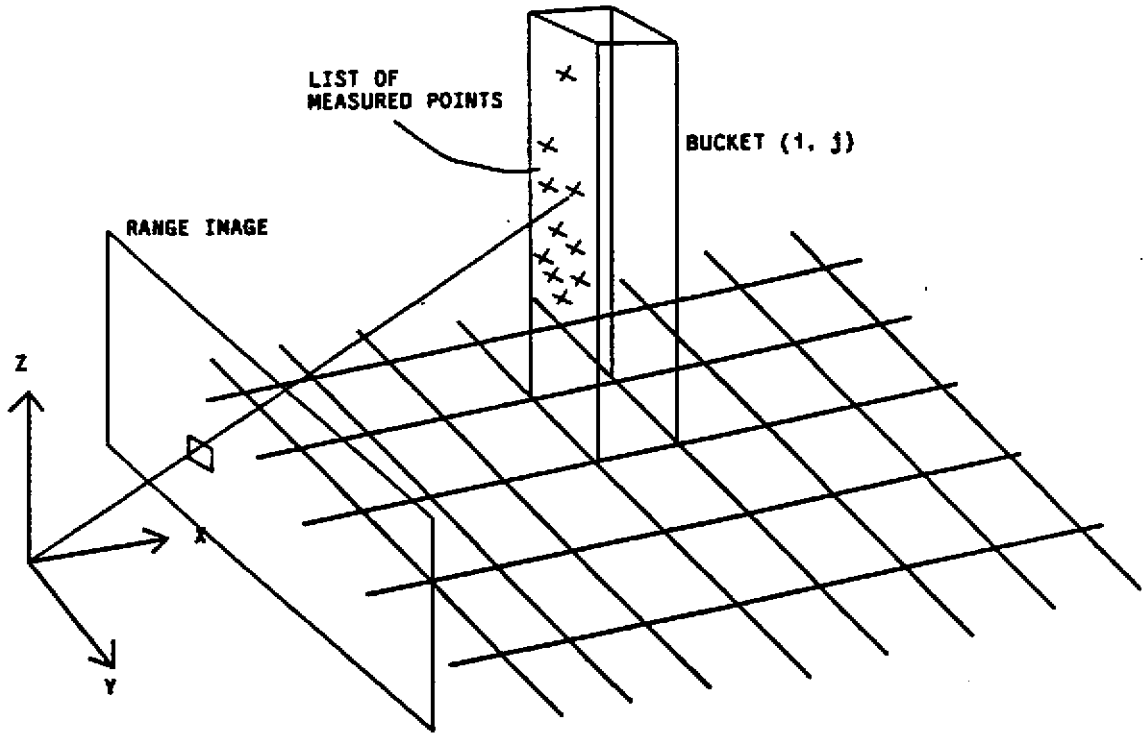


Figure 2.7: Building the obstacle map. 3-D data points are projected into discrete buckets on a horizontal grid.

The obstacle detection algorithm does not make assumptions on the position of the ground plane, in that it only assumes that the plane is roughly horizontal with respect to the vehicle. Computing the slopes within each cell has a smoothing effect that may cause real obstacles to be undetected. Therefore, the resolution of the elevation map must be chosen so that each cell is significantly smaller than the typical expected obstacles. In the case of Figure 2.8, the resolution is twenty centimeters. The size of the detectable obstacle also varies with the distance from the vehicle due to sparser range pixels at longer distances.

In fast obstacle detection mode, several improvements can be used to decrease the computation time. We need to look for obstacles only in a narrow stripe in front of the vehicle. We do not need to detect all the objects, it is sufficient to raise an alarm as soon as one object is found. We do not need high spatial resolution at close range, therefore the data can be subsampled close to the vehicle. Taking into account those improvements, we achieved fast obstacle detection that runs in 600 ms on a SPARC workstation, which is fast enough at the current speeds, considering the fact that the acquisition time is still 500 ms on average.

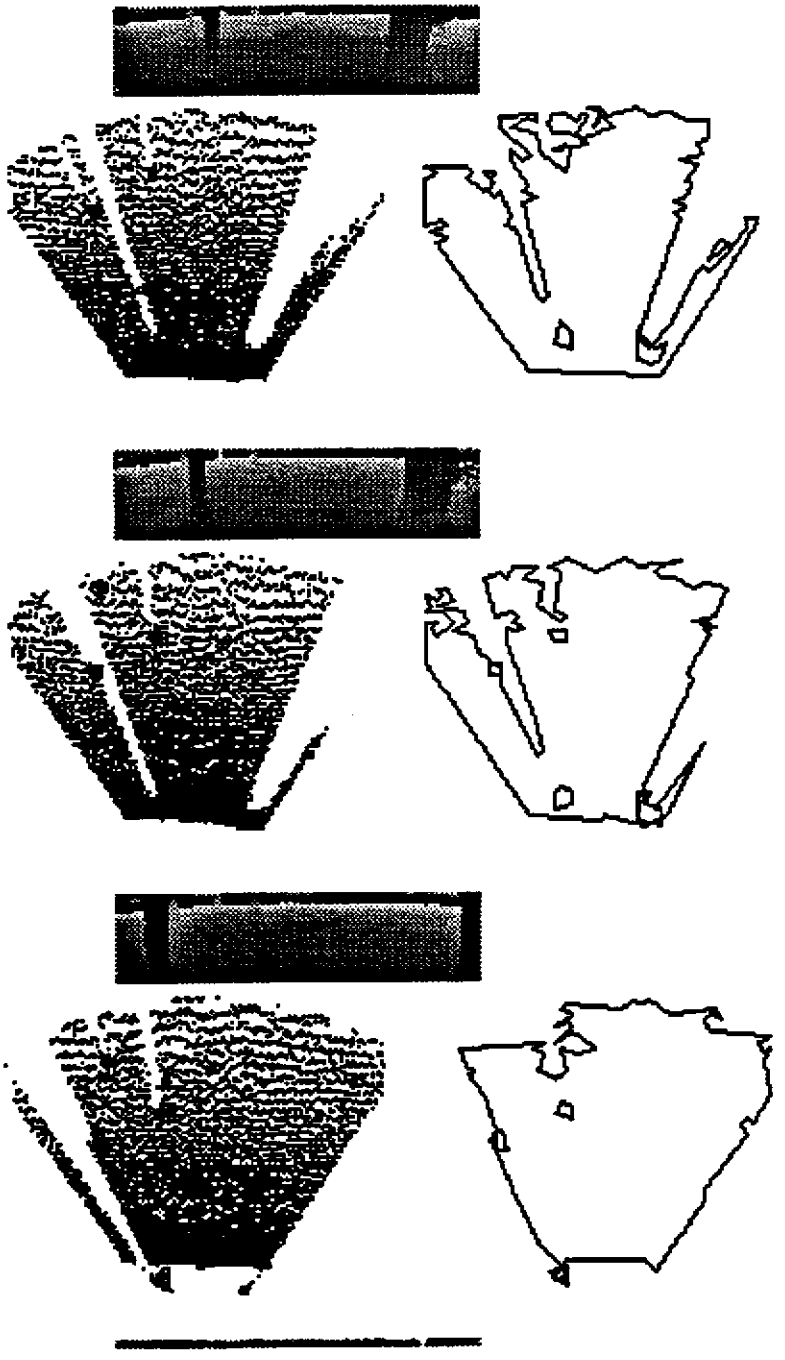


Figure 2.8: Obstacle detection on a sequence of images. For each image, top: original range image; bottom left: overhead view; bottom right: segmented elevation map.

Another application of object detection is to build object maps by combining many observations. Combining observations is critical to improve object localization and to remove spurious objects. Matching objects is not very expensive in our case because we have only a few objects to match in each frame and because we can assume that we have a reasonable estimate of the displacement between frames from INS or dead-reckoning so that the locations of the objects detected in one image can be easily predicted in the next image. The main issues are to remove spurious objects and to compute the location of the objects as accurately as possible. Spurious objects can be detected in two cases: noise in the range image may cause the object detection program to hallucinate, and moving

objects (*e.g.* people) crossing the field of view are detected as objects even though they should not be included in a map. Spurious objects must be eliminated because they may lead to disastrous results when they are used later on to correct the position of the vehicle. The position of the objects must be computed as accurately as possible so that the position corrections that are computed using the object map are also accurate.

The problem of spurious objects is solved by calculating a confidence measure for each object. Once an object has been seen in one image, it should appear in subsequent images, as predicted by vehicle motion, object position, and sensor field of view. If it appears as predicted, its confidence is increased, otherwise its confidence decreases. Objects with low confidence are discarded. Accurate object locations are obtained by updating the uncertainty on object location (mean and covariance matrix) each time an object is observed in a new image. The initial uncertainty is based on a sensor model and depends mostly on the distance between the object and the vehicle. The uncertainty also takes into account the fact that only a small part of the object surface is observed. The uncertainties are combined using standard maximum likelihood techniques. Figure 2.9 shows a sequence of fourteen images. The images are separated by about 50 cm. The white lines connect the objects that are matched between images. The white dots indicate the locations of the detected objects in the images. Spurious objects are detected in images 13, 18, 20, and 22. Since they are not matched, their confidence is low and they are eventually discarded from the map.

2.3.3 Terrain modeling for cross-country navigation

Obstacle detection is sufficient for navigation in flat terrain with discrete obstacles, such as following a road bordered by trees. We need a more detailed description when the terrain is uneven as in the case of cross-country navigation. For that purpose, an elevation map could be used directly [12] by a path planner. This approach is costly because of the amount of data to be handled by the planner which does not need such a high resolution description to do the job in most cases. For example, the planner should not need to scan a full elevation map if the terrain is completely flat. In this example, the terrain representation should provide enough information to quickly identify the fact that no search is needed. An alternative to elevation maps is to group smooth portions of the terrain into regions and edges that are the basic units manipulated by the planner. This set of features provides a compact terrain representation. However, the planner may still need information at a higher resolution than the feature map. For example, in cluttered environments the planner has to examine small portions of the terrain to decide which areas are traversable [39]. Therefore, a compromise representation should include both high resolution elevation data and feature information and should allow for efficient access to large chunks of terrain.

Such a compromise is realized by organizing elevation and feature maps in a quadtree structure. Each node of the tree contains information that describes the portion of the terrain covered by the corresponding quadrant: minimum and maximum elevation, maximum slope, average elevation, and maximum discontinuity within the quadrant. Discontinuities and slopes are computed by applying a gradient operator to the elevation map. Using this representation saves a considerable amount of computation time both in building the terrain representation and in using it for path planning. A complete terrain representation can now be built in 2 seconds on a SPARC workstation. Figure 2.11 shows several levels of the quadtree representation built from the range image of Figure 2.10.

The terrain model from a single range image may not be sufficient due to the limited field of view of the sensor. In our case, the map is accurate enough up to 6 meters in front of the vehicle. However, quadtree representations from consecutive images can be fused to yield a larger model of the terrain. In the current Navlab configuration, we use the INS readings to register in x, y, roll, pitch, and yaw. Registration in z is achieved by calculating the z offset between maps as the mean difference of z values.

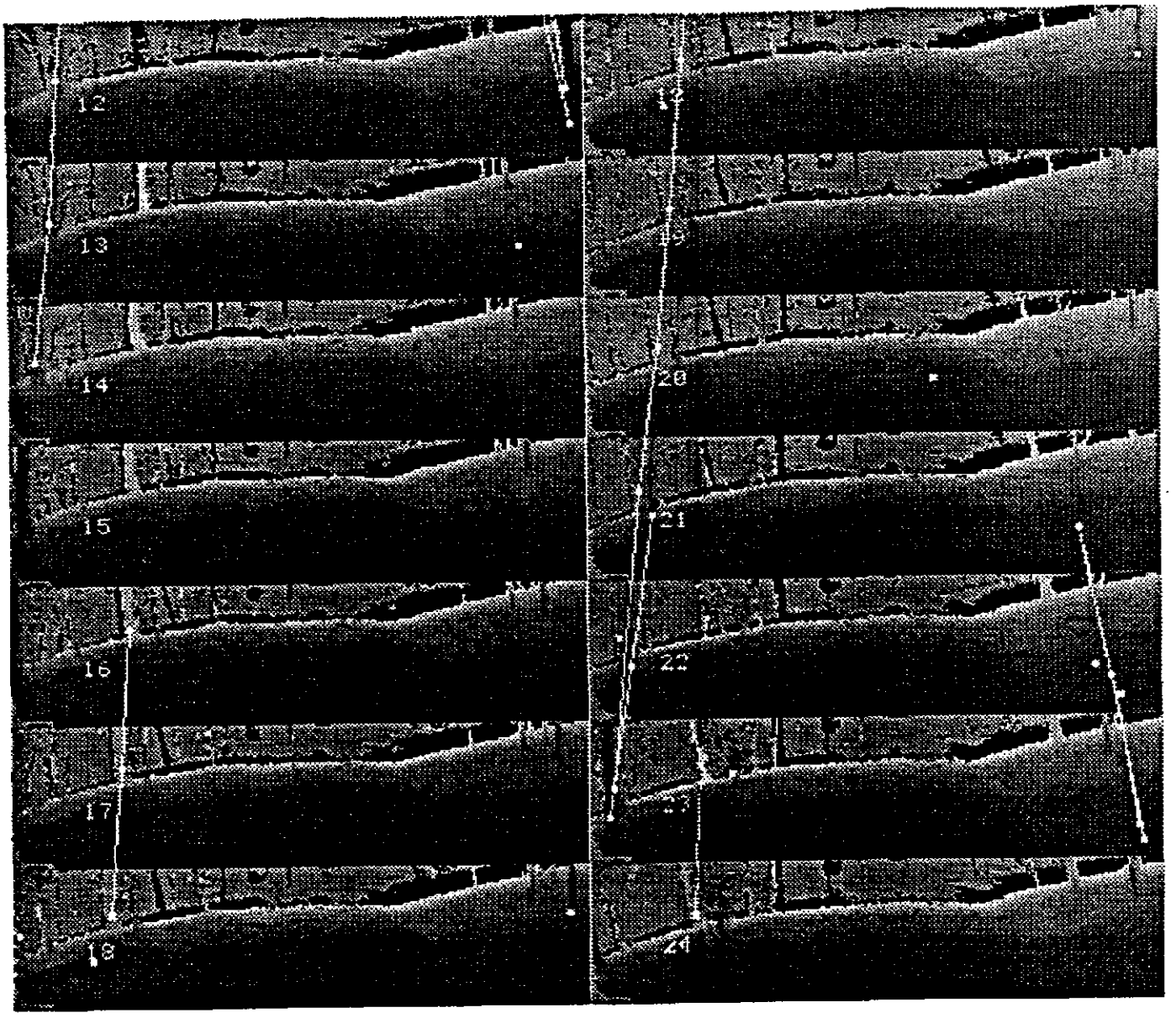


Figure 2.9: Matching objects in a sequence of range images. White lines show corresponding objects in sequential images.

Our goal was to provide our cross country planner with a method of getting information about the terrain around it efficiently. Without some intermediate data processing, the planner would have to search individual elements of the elevation map to obtain the information it needs. Since the planner examines many overlapping areas, using this method causes much duplication of effort.

There is no apparent compact and elegant method to model the terrain that a cross country system needs to traverse. For one thing, unlike in road following, there are no regions that are guaranteed to be traversable. Unlike indoor navigation, the cross-country system has to deal with arbitrarily shaped regions of untraversable terrain, not

just discrete objects. In addition, due to the complexity and randomness of natural terrain, any attempt to make the problem simpler by fitting mathematical models to the terrain ends up with a representation that is just as intractable as an elevation map.

Instead of looking for a more compact representation, we achieved our goal by implementing a system that represented the terrain hierarchically, in a pyramid of elevation maps of descending resolution. Getting information about a patch of terrain using the terrain pyramid is more efficient than using the raw elevation map in the same way that representing an object with a quad-tree is more efficient that representing it with pixels. For a small initial cost, the amount of duplicated computational effort needed by the cross country planner is vastly reduced.

By considering what the planner needed to know about the terrain, we were able to reduce the computations by adding extra features to the terrain pyramid, such as terrain discontinuity and terrain gradient. We only added such features if the initial overhead of calculating the feature was less then the computational benefit to the planner.

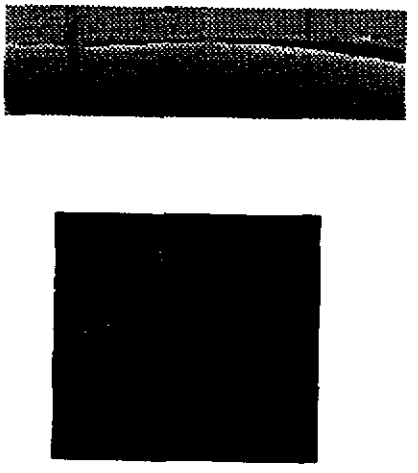


Figure 2.10: Range image and elevation map.

2.3.4 Map building from terrain features

As in the case of object descriptions, composite maps can be built from terrain descriptions. The basic problem is to match terrain features between successive images and to compute the transformation between features. In this case the features are the regions that describe the terrain parameterized by their areas, the equation of the underlying surface, the center of the region, and the main directions of the region. If objects are detected they are also used in the matching in the same way as before. Finally, if the vehicle is traveling on a road, the edges of the road can also be used for the matching. As in the case of object matching, an initial estimate of the displacement between successive frames is used to predict the matching features. A search procedure is used to find the most consistent set of matches. As before, the search is actually very fast due to the small number of features and the fact that the initial guess of the transformation between images is usually quite close to actual value. The features are weighted in the search according to how reliably they can be detected. The reliability of a feature depends on its type: discrete objects are more reliable than terrain regions and road edges. Once a set of consistent matches is found, the transformation between frames is recomputed and the common features are merged.

This map building approach has been tested on sequences of images with errors in position estimation of up to 1

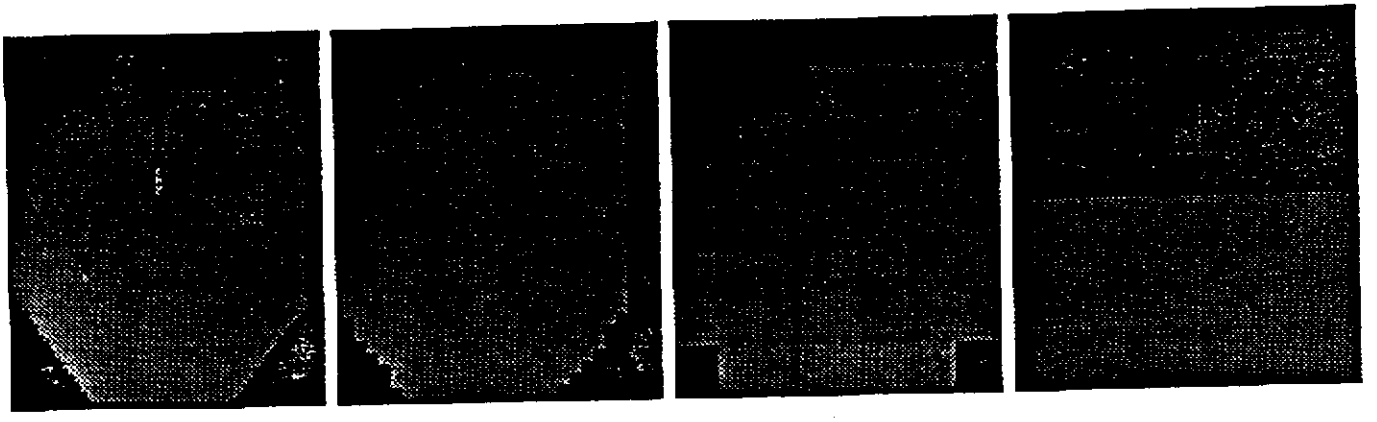


Figure 2.11: Four levels of the terrain quadtree.

meter in translation and 20 degrees in rotation. For example, figure 2.12 shows a sequence of five maps that are merged into a composite map using feature matching.

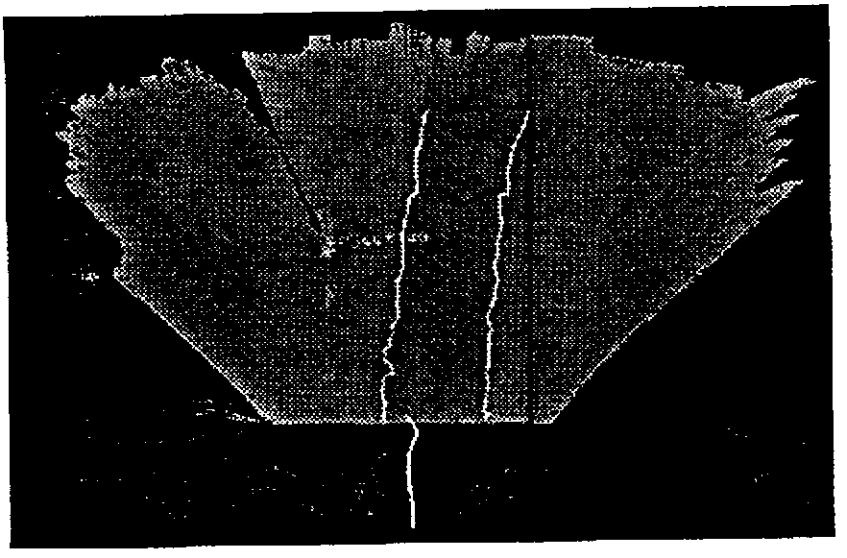


Figure 2.12: 3-D map built from 5 range images.

2.3.5 High resolution terrain models

The high resolution terrain representation is an elevation map, that is a function $z=f(x,y)$ represented by a regular grid of values (x_i,y_i). The most straightforward way to convert a range image to an elevation map representation would be to map each pixel (*row,column,range*) of the range image to an (x,y,z) location in map coordinates. There are a number of problems with this approach:

- *Sampling:* Since this approach is similar to image warping, the distribution of data points in the elevation map is not uniform. The map gets sparser farther from the sensor.
- *Shadows:* Objects create range shadows, that is regions of space that are not visible even though they lie within the sensor's field of view. Shadowed regions must be explicitly identified and represented separately since no information is available in those regions. The difficulty here is to distinguish between genuine range shadows and regions of the map with no information due to sparse sampling.

- *Uncertainty:* Range measurements are corrupted by noise due to electronic noise, surface material, laser footprint, etc. The noise can be modeled by a variance, σ, for each measurement. As a first approximation, σ depends only on the measured range. This uncertainty is represented in sensor space and must be converted into a representation of the uncertainty in the elevation map.

Those problems could be solved by applying a standard interpolation technique to the sparse elevation map. This would provide a dense elevation that is a reasonable interpolation of the sparse input. However, such an interpolation technique would not take into account the geometry of the sensor thus making it difficult to identify shadows or to convert sensor uncertainty to map uncertainty.

The *locus* algorithm overcomes many of these problems by explicitly taking into account the sensor geometry in building a dense elevation map. The idea is illustrated in Figure 2.13: Finding the elevation z of a point (x,y) is equivalent to computing the intersection of the surface observed by the sensor with a vertical line passing through (x,y). Knowing the geometry of the sensor, the line can be represented in image space by an analytical equation of the form $range=f(r,c)$ where r and c are the row and column coordinates in the image. (The projection of this line into the image defines a locus of points which gives the algorithm its name.) The intersection between the line and the observed surface is found between two adjacent pixels (r_1,c_1) and (r_2,c_2) such that $range_1<f(r_1,c_1)$ and $range_2>f(r_2,c_2)$, where $range_1$ and $range_2$ are the values in the image. The final value of z is obtained by interpolating the range between (r_1,c_1) and (r_2,c_2).

The key point of the locus algorithm is that the interpolation is taking place in the *image* instead of in the map. This allows us to explicitly take into account the sensor model: the uncertainty on z is computed by combining the known uncertainties at (r_1,c_1) and (r_2,c_2). The unknown regions in the map can be detected by observing that (x,y) belongs to an unknown region of the map if (r_1,c_1) or (r_2,c_2) are on a range discontinuity in the image. Another important consequence is that the elevation can be computed at any point of the map without having to recompute the entire map whereas standard map interpolation would have to compute the entire sparse map before interpolating the dense map. Finally, there is no constraint on where the map coordinate system is located with respect to the image. In particular, we can generalize the algorithm to compute the intersection of any line in space with the image.

This technique has been used to build terrain maps, with resolutions as fine as 10 cm, that include uncertainty and explicit representation of unknown regions. The locus algorithm can also be used to build large maps by matching maps from individual images. Two images of the same area taken from two different locations, are related by a transformation T (rotation and translation) between the two locations. The matching problem is essentially to compute T as accurately as possible. Once this is done, the maps can easily be merged into a larger composite map. Given some value of T, we can compute from the images two elevations z_1 and z_2 for each point (x,y) in the map. The squared difference $(z_1-z_2)^2$ is a measure of how good our knowledge of T is. Since one point in the map is not sufficient because of the uncertainty, we can use the sum of the squared differences, $E(T)$, over the part of the map that is visible in both images. $E(T)$ is minimum when T is the exact transformation between the locations at which the two images were taken. Starting with an initial estimation of T we can therefore apply a minimization algorithm (gradient descent) to $E(T)$ to compute the best possible value of T. In practice, the initial value of T is computed from feature matching or from the positioning system of the vehicle. $E(T)$ is not computed over smooth areas of the map, which would provide little variation as a function of T.

This matching technique has been applied to the building of large maps (several hundred meters) using many range images collected as the vehicle travels. Experimental results show that the locus algorithm can be used to build accurate maps over long distances of travel. Figure 2.14 shows a map built by combining 122 range images.

The locus algorithm provides a basis for a number of other map operations. For instance, matching local maps for

the vehicle with low-resolution aerial elevation maps can be implemented using the locus algorithm. Detailed terrain features, such as ridges and valleys can be extracted from high resolution maps [27].

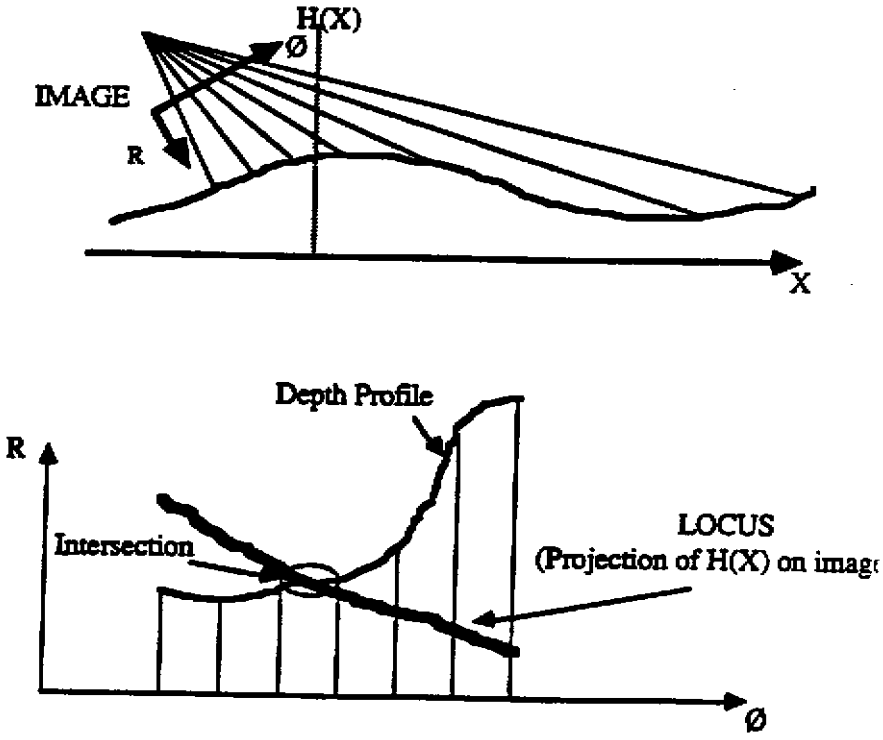


Figure 2.13: The Locus Method: intersection of scanned surface with vertical line in world space (top), and same intersection in image space (bottom).

2.3.6 Discussion

The terrain representations that we have developed have proved to be critical in building a successful mobile robot that includes capabilities of obstacle avoidance, open terrain navigation, and map building. We have also demonstrated that laser range finding is a sensor modality that should be used, being superior to passive techniques at least at this stage of the research. There are however a number of additional problems, including:

- *Sensor fusion:* Using geometric information is sufficient for most navigation tasks. In other areas it would be beneficial to explicitly merge geometric information with appearance information. For example, shape and color are equally important in object recognition. In road following, the use of geometric information would help distinguish between shadows and other illumination effects and the presence of real objects on the road. This would greatly improve the performance of color classification but requires that color and range information be merged. We have done some limited experiments with sensor fusion both at the level of the images, constructing a "colored-range" image, and at the level of the features extracted from color and range images. The main issues are the choice of the level at which information should be merged (images, features, or interpretation), and the difficulty of accurately registering range sensor and color cameras. Much more work remains to be done in this area.
- *Use of reflectance:* In addition to range, a laser scanner can also measure an image of the energy of the reflected laser beam. This image, usually called the *reflectance* image, is similar to an intensity image

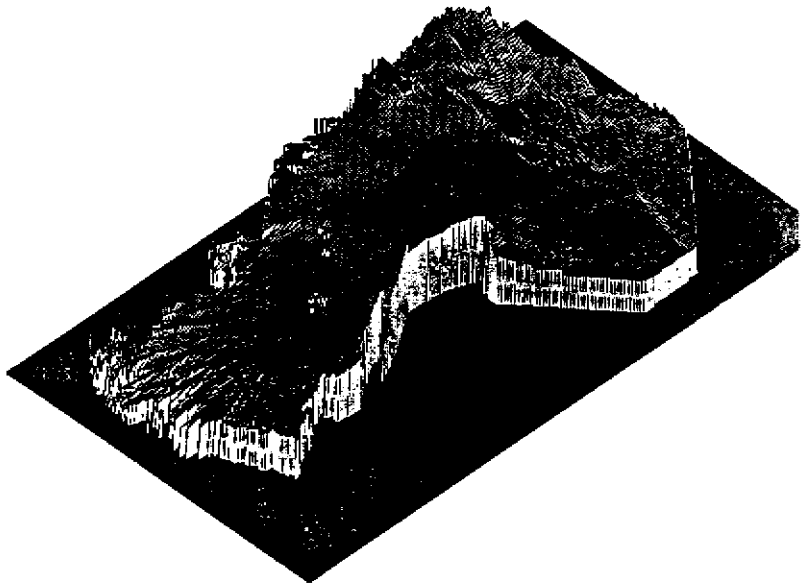


Figure 2.14: An elevation map built by the locus algorithm from 122 range images, covering 250 meters.

except that it is largely insensitive to outside illumination. Therefore it does not exhibit effects such as shadows, highlights, or interreflection, all of which are hard to model. We have used reflectance data for road following and object recognition with some success. The main limitation was poor performance of the reflectance measuring, which we believe is not inherent to the technology but is due to the particular sensor that we were using. Those preliminary experiments have shown that active reflectance images are an attractive alternative to intensity images and that research in this direction should be pursued further.

- *Uncertainty:* We have proposed ways of representing uncertainty for various terrain representations. However, the way we deal with uncertainty is still somewhat ad hoc in that it makes heavy use of the characteristics of our sensor and the particular representations that we have developed. A more systematic approach to modeling uncertainty in 3-D terrain is needed.

2.4 Planning

Intelligent action requires perception, planning, and control. While our main emphasis has been on perception, we have also developed the planning and control needed for smoothly following roads, and for traversing rugged off-road terrain that challenges the limits of the vehicle hardware.

The role of planning is to generate trajectories that meet goal requirements (such as positioning to see a landmark, for example) without endangering the robot. It must also make sure that the robot is kinematically able to execute these trajectories, all in the presence of uncertainty in the robot's control and environment. A number of systems have been built that address a subset of these issues. Early planners were targeted for indoor mobile robots and assumed that the environment could be modeled as a flat surface with polygonal or polyhedral objects [9, 24, 30]. Furthermore, the robot was assumed to be circular and omnidirectional. Later, Laumond [28] and Jacobs [18] relaxed the omnidirectional requirement by modeling a car-like robot with a minimum turning radius. In a system developed at Hughes [12], the indoor environment constraint was relaxed and a planner was developed to plan paths in off-road environments. None of the above systems were able to reason about sophisticated goal requirements and uncertainty in perception and control. The theoretical groundwork was spelled out in [38], but without actually building a usable planner. Planning with uncertainty has been explored in manipulation [31], but it is unclear how to apply the domain-specific nature of these techniques to planning for mobile robots.

While the contributions of the above work are very important, in many cases it is difficult to see how to extend them to address the remaining issues or how to generalize to other robots or environments. Our planner addresses those problems by providing a framework for building efficient planners for different types of robots, environments, goals, and uncertainty models.

The first step in building a planner is to define the constraints that must be used to compute a safe trajectory and to reach the goal. We define three types of constraints: sensing, environmental, and kinematic. First, we select positions at which the robot registers its position relative to the world or to map new areas. Those positions are intermediate goals that provide additional constraints to the planner. Second, we identify placements (configurations) of the robot in the environment that will incapacitate it or render it unable to locomote. They define environmental constraints that a trajectory must satisfy. Such configurations include those that bring the robot in contact with other objects in the environment, as has been modeled in traditional indoor robotics. Outdoor robots face other hazards as well. Configurations that cause the robot to tip over or place it in situations where it cannot propel itself forward are also be avoided. Figure 2.15 shows a set of environmental constraints. Third, we define kinematic constraints. Most robots are not omnidirectional. They cannot travel between two arbitrary configurations within given bounds. For example, car-like vehicles cannot translate directly sideways. In the case of the Navlab, the minimum turning radius is seven meters. In addition to the three basic constraints, uncertainty in robot position must be taken into account. Sources of uncertainty range from random error in the robot's control to gross errors such as wheel slippage. Our local path planner accounts for control-based uncertainty to avoid collisions and to guarantee goal attainment.

The planner generates trajectories to the next sensing point using a range map of the terrain in front of the robot acquired from an ERIM laser rangefinder. Since the Navlab's pose can be represented by two translational parameters and one heading parameter, the planner must find an admissible trajectory through a three-dimensional configuration space. Conceptually, each constraint is represented by a functional inequality of the form $f(p) < K$, where p is the vector of robot configuration parameters. The constraint is satisfied if the inequality is satisfied. Applying the constraints divides the configuration space into admissible subspaces. The sensing positions form a subspace of this configuration space which comprises the goal of the path. The environmental constraints form a subspace representing unsafe configurations for the robot. The kinematic constraints dictate the functional form of the trajectory, and the uncertainty constraints dictate an envelope about the trajectory guaranteed to contain the robot.

Analytic approaches to the problem are infeasible given the complexity of some of the constraint functions. Furthermore, the constraints are dependent on the terrain itself, which does not have a functional form. A straightforward approach is to tesselate the space into pixel-sized points, evaluate the constraints at each point, and search the resultant lattice. However, even for moderately-sized planning spaces, the number of points (states) makes the search prohibitively expensive. Instead, our planner finds paths for a mobile robot using a parameter resolution hierarchy. In this hierarchy, all constraints (sensing, environmental, kinematic, uncertainty, etc.) are evaluated across a subspace of configurations at a time (rather than individual configuration points), thus reducing the total number of states in the search. Sensing and environmental constraints are evaluated across three-dimensional voxels in configuration space, kinematic constraints are enforced between faces of the voxels, and uncertainty constraints determine the voxel expansion needed to bound the robot's pose. The planner finds a trajectory by searching connected sequences of voxels. For a given subspace, the planner evaluates each constraint to determine whether all, none, or some configurations in the subspace satisfy the constraint. The planner begins by considering large subspaces. Passage through the subspace is permitted if all constraints are satisfied for all configurations. If at least one constraint fails for all configurations, the entire subspace is untraversable and is removed from further consideration. In the event of the remaining case (at least one constraint is not satisfied by at least one configuration), the subspace may be traversable, so the planner subdivides the subspace into smaller spaces

and continues to plan at a higher resolution. Most of the constraints are modeled uniformly as functional inequalities. Thus, the planner can classify a subspace into one of the three cases by computing the upper and lower bounds for the function across the subspace and comparing them to a constant. A cost can be assigned to each subspace, and a standard, depth-first, breadth-first, or heuristic search can be employed.

The quadtree representation of the terrain described above provides an efficient way to implement the hierarchical search. Relevant terrain parameters such as min and max elevation are maintained for each quadrant so that the planner does not have to go back to the highest resolution map to evaluate its constraints. Figure 2.16 shows one slice of the constraint space generated by running our cross-country planner on the elevation map of Figure 2.10. The crossed areas represent inadmissible areas of terrain, i.e., areas on which it is illegal to place the center of the vehicle. In the process of planning this path, the planner made 1493 queries for terrain information. This shows the efficiency of both hierarchical search and hierarchical representation of the terrain.

Due to the uniform way in which the constraints are modeled and the resolution hierarchy is built, the framework employed in this planner is applicable to other classes of robots, environments, and goal specifications. Future work will include building a complete system around the planner to autonomously drive the Navlab off-road, implementing algorithmic improvements and utilizing faster computer hardware to increase performance, and extending this work to operate on more capable off-road vehicles.

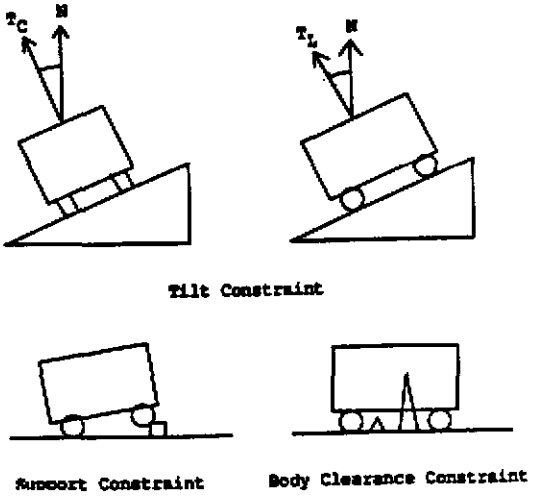


Figure 2.15: Environmental constraints.

2.5 Architectures and Systems

The software architecture of a mobile robot is the framework that assembles the separate components, for sensing, planning, and control, into a coherent system. Simple robots, performing simple tasks, often have an "architecture" that consists of a fixed sequence of subroutine calls, repeated without variation. More complex robots and missions require more structure, to enable changing behaviors and conflicting subgoals, and to specify functions and interfaces so groups of researchers can contribute to building the system.

The current architecture for the Navlab is based on a toolkit called EDDIE, which provides communications and a tight interface to our low-level vehicle control. On top of EDDIE, we have built tools such as the Annotated Map, a mechanism for storing object and mission information. Our most ambitious systems have used EDDIE and annotated maps for navigating suburban streets, in a system we call the "Autonomous Mail Vehicle".

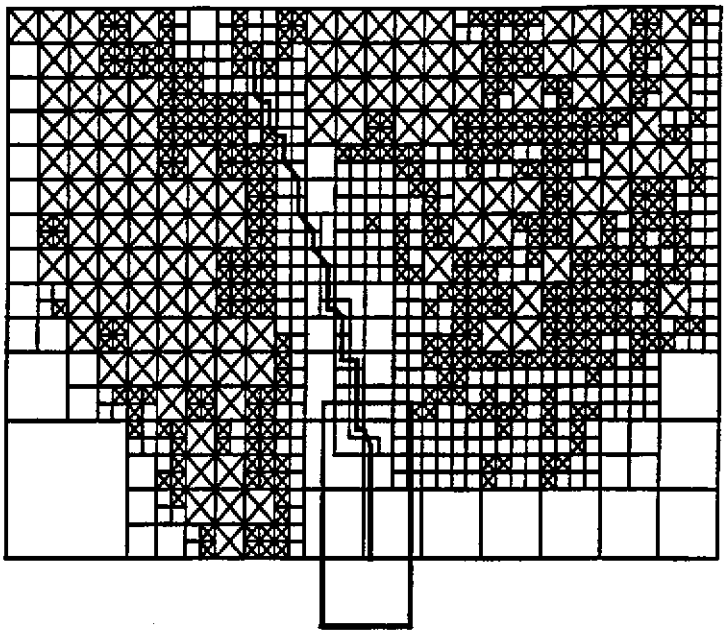


Figure 2.16: Planned path through cross-country terrain. Crossed squares are inadmissible regions, passable areas are empty squares.

2.5.1 Background

The most conventional architectures separate robot software into separate modules for sensing, thinking, and control. This has the advantage of giving one module control of the vehicle, another control of all sensors, and a third control of modeling and planning. This decomposition groups design tasks in the likely areas of expertise of separate research groups. The drawback of this approach is that it does not allow for high-speed special-purpose reflexes, that must do sensing, thinking, and control all in one tightly-integrated module.

The opposite approach is typified by Brooks in his subsumption architecture [8]. In his robots, each module covers the complete range from sensory input to control output. He divides his modules into a hierarchy of functions, each "subsuming" the lower levels. The first module watches sensor data and moves the vehicle away from obstacles. The next layer moves the vehicle randomly, unless the lowest layer takes over to avoid hitting an object. Higher layers add purpose to the wandering (e.g. towards open doorways), look for objects of interest, and so forth. Each layer is relatively simple to build, and at least in principle mostly decoupled from adjacent layers. But with no central world model, it takes careful design to ensure that various modules are not working at cross purposes. Related ideas include reactive or reflexive planning, which emphasize quick response rather than careful preplanning; and behaviors, which package sensing and control modes appropriate for specific situations [34].

Several attempts have been made to build architectures that combine the best of both approaches. These systems typically propose a hierarchy, in which sensor interpretation at each level feeds into both planning at the same level, and higher-level sensor interpretation [1]. Plans at each level are decomposed into lower-level steps, and given to the next lower level for execution. The hierarchies are often structured by time (quick reflexes at the low level, through slower processes at higher levels); data abstraction (raw signals to symbolic reasoning); and space (local effects to global databases). In trying to encompass all possible systems, these general-purpose architectures lose their prescriptive power. Their main contribution may instead be descriptive, providing a common vocabulary in which to discuss the differences between architectures.

For the Navlab, our first real architecture was CODGER, for COmmunications Database for GEometric Reasoning [16, 37, 40]. CODGER is a centralized architecture, focused on a module called the Local Map Builder

(LMB). CODGER was designed to handle all communications and geometric transforms, to make it easier to build and interface individual modules. Communications are anonymous; modules send data and requests to the LMB, and receive responses when available, without knowing which other modules are generating or using data or where those modules are running. The LMB stores all geometric objects, and keeps track of a history list of vehicle motion and position updates. CODGER uses its history list to answer geometric queries that involved multiple coordinate frames. The LMB can take a location specified relative to the vehicle at a particular time, and return the coordinates of that point, either in the world frame, or relative to the vehicle at a different time.

2.5.2 EDDIE

Our current system on the Navlab is based on EDDIE (Efficient Decentralized Database and Interface Experiment). EDDIE does not specify a particular architecture, but rather provides a toolkit which allows specific systems to be built quickly and easily. Vehicle positions are maintained by the lowest level controller, which has the closest access to the vehicle and therefore the most accurate information. Communications are greatly simplified, and are point to point, increasing their efficiency. The map is divided into local and global representations. By splitting architectural functions into separate pieces for local communications, vehicle history, and map handling, the individual modules are much smaller and easier to maintain.

The first part of EDDIE is the new real-time controller. This module does low-level vehicle control and handles communication with higher-level modules, and in addition maintains the current vehicle position. Vehicle motion commands arrive at the controller labeled as either "immediate" or "queued". The controller parses incoming commands, handles the queue, and talks to the hardware motion controller at the appropriate times to set new steering wheel positions and vehicle velocities. By querying the vehicle's encoders at frequent intervals, the controller is able to maintain an accurate dead-reckoned position estimate. In EDDIE, no vehicle position history is kept. The only times when it is necessary to know vehicle position are when new data is acquired, or during trajectory planning. It is easier, and more accurate, to dispense with history mechanisms, and instead to query the controller for the current vehicle position each time an image is digitized, and whenever a planner needs to know the vehicle's location.

The vehicle controller uses different tracking strategies to keep the vehicle on the desired path. It can also be called upon to follow a previously recorded map if the perception clients are temporarily unable to navigate the vehicle. This keeps the vehicle on a safe path while the vehicle turns sharp corners, outside the camera's field of view, or travels through featureless or confusing visual scenes. Another safety consideration is smoothly regulating velocity, trading some reduction in accuracy of velocity for smooth accelerations and reduced vehicle roll around sharp curves. The controller warns against system failures and records a log of events for future reference. This is extremely valuable in system configuration and debugging. The low-level controller is also responsible for utilizing INS and encoder data to find the best estimate of current position and relaying it to external clients through the ethernet. Figure 2.17 shows accurate vehicle position estimation, using the INS (solid line), and the less accurate, but still stable, estimation using only dead reckoning (dotted line). The clients are managed by a software server which prioritizes the connections in order to meet the needs of many clients without degrading the level of performance required by critical components of the system [2].

Closing all position-estimation loops through the controller allows transparent path modifications. We have implemented a joystick interface that allows a user to modify commanded trajectories. Joystick input is simply summed with computer input, so the user has the sensation of "nudging" the vehicle away from its planned path. The Navlab is also being equipped with a "soft bumper", a ring of ultrasonic range sensors to detect nearby objects before collision. When completed, the soft bumper will interact with the controller in the same manner as the joystick, by adding its control input to the input from planning, but will have progressively higher gains as the time

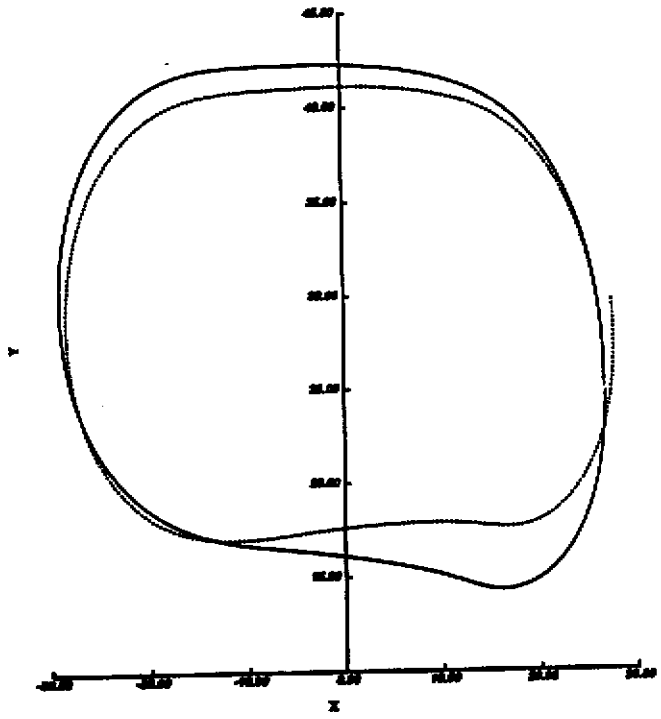


Figure 2.17: Position estimation during a robot run. The solid line shows the accurate vehicle track given by inertial navigation sensors. The dotted line shows the less accurate vehicle track estimated by dead reckoning.

to collision decreases. Previous systems would have been destroyed by this subversion of planned paths, since CODGER kept vehicle position history by an open-loop expectation of perfect path tracking. In the EDDIE system, all position queries are handled directly by the controller, and are therefore answered correctly even if the path has been modified.

Communications in EDDIE are unexotic and uninteresting, but fast, with point-to-point connections. We currently use TCP/IP over the ethernet, but could go to shared memory or other protocols for particular connections, as needed. Instead of building special-purpose synchronization mechanisms, EDDIE simply uses a blocking read to pause module execution until data arrives.

2.5.3 Annotated Maps

EDDIE does not have a global map at the center. Local positions, used only for the purposes of obstacle avoidance or path following, are never written into a map. Global, permanent, maps are handled by the separate mechanism of "annotated maps".

Annotated maps start with a geometric representation of objects, such as roads, intersections, and landmarks. Annotations store additional information, not usually contained in maps, tied to a particular location or object. Annotations hold a wide variety of knowledge, both procedural (actions and methods) and declarative (data), tied to a particular map location or object. Annotations can range from high-level ("church") to geometric ("steeple height 25m, ...") to sensor-specific ("look for long nearly-vertical edges") to raw data ("color R1 G1 B1"). The knowledge in an annotation can come from a wide variety of sources, such as human experts, mission planning software, and even the vehicle's own observations and experiences on previous missions.

A map manager module controls the annotated map. Two forms of access are provided, queries and triggers. Queries allow a module to fetch information on demand. They return all annotations of the requested type within a specified polygon. Typical queries ask for descriptions of landmarks, or for which recognition methods have worked for this landmark on previous vehicle runs. Triggers are a special form of annotations, monitored by the EDDIE map manager. When the vehicle reaches the trigger's location, the map manager automatically sends a specified message to a named module. Triggers may be set up during mission planning, and used to wake up sleeping processes at specified locations or to alert a running module to a change in conditions. In a typical run, triggers are used to tell the vehicle when and where to look for landmarks, and when to switch from straight road following to the slower intersection navigation code.

Annotated maps are not designed to be a master control, but rather to serve as a scratchpad (for queries) and alarm clock (for triggers), in the EDDIE architecture. Annotations have a standard format for header information, such as type and location. The format for the rest of the annotation is defined by the modules that post and retrieve the annotations, and need not be interpreted by the map manager.

Annotated maps provide a convenient framework for organizing knowledge. Tying the knowledge in annotations to particular locations in the map makes it possible to pre-plan difficult mission segments, and to retrieve that information efficiently during execution. This framework enables missions that would not otherwise be possible, due to real-time constraints and limits in processing and algorithmic power.

2.5.4 AMV

We have built several systems on top of EDDIE and the Annotated Maps. The road following system for the Navlab is the Autonomous Mail Vehicle, or AMV. This system draws its inspiration from postal deliveries in suburban or rural areas, which follow the same route day after day, undeterred by "rain nor snow nor dark of stormy night". The mail carriers drive at relatively slow speeds, often on many different kinds of roads. They do gross navigation through a network of roads and intersections, and fine position servoing to mail boxes.

This type of system is an example of a broader class of applications which focus on map building and reuse, positioning, road following, and object recognition. Our AMV project is investigating those issues, including strategies for using different sensors and different image understanding operators for the perception components.

The most ambitious mission we have performed to date is a 0.4 mile run on unmodified suburban streets in Pittsburgh's North Hills. This involved:

- Driving along curving suburban streets, with no pavement markings, including many different types of driveways;
- Traversing four intersections, at two of which the Navlab had to make a 90 degree left turn;
- Stopping for unexpected obstacles, and resuming motion when clear;
- Locating landmarks for position updates and for finding the destination.

We built an annotated map of the route, driving the Navlab by hand and using the laser scanner to record the location of 3-D objects. Object positions were measured in multiple images, to discard moving objects (pedestrians, cars, dogs) and to improve the accuracy of measured position. The map was then annotated with triggers that controlled vehicle path execution. During the run, the vehicle started moving slowly, while it found landmarks to initialize its position. A trigger then caused the vehicle to speed up until it approached the first turn. At that point, triggers caused various modules to slow the Navlab, find 3-D objects, match them against the map, and update the vehicle's position estimate. Through the turn, vision was not able to see the road, so another trigger caused dead reckoning to take control until the vehicle was lined up with the next road, when the road was again in the field of view and vision could resume control. The run proceeded in this fashion until the final triggers, which matched the mailbox at the destination with the map, and brought the vehicle to a stop.

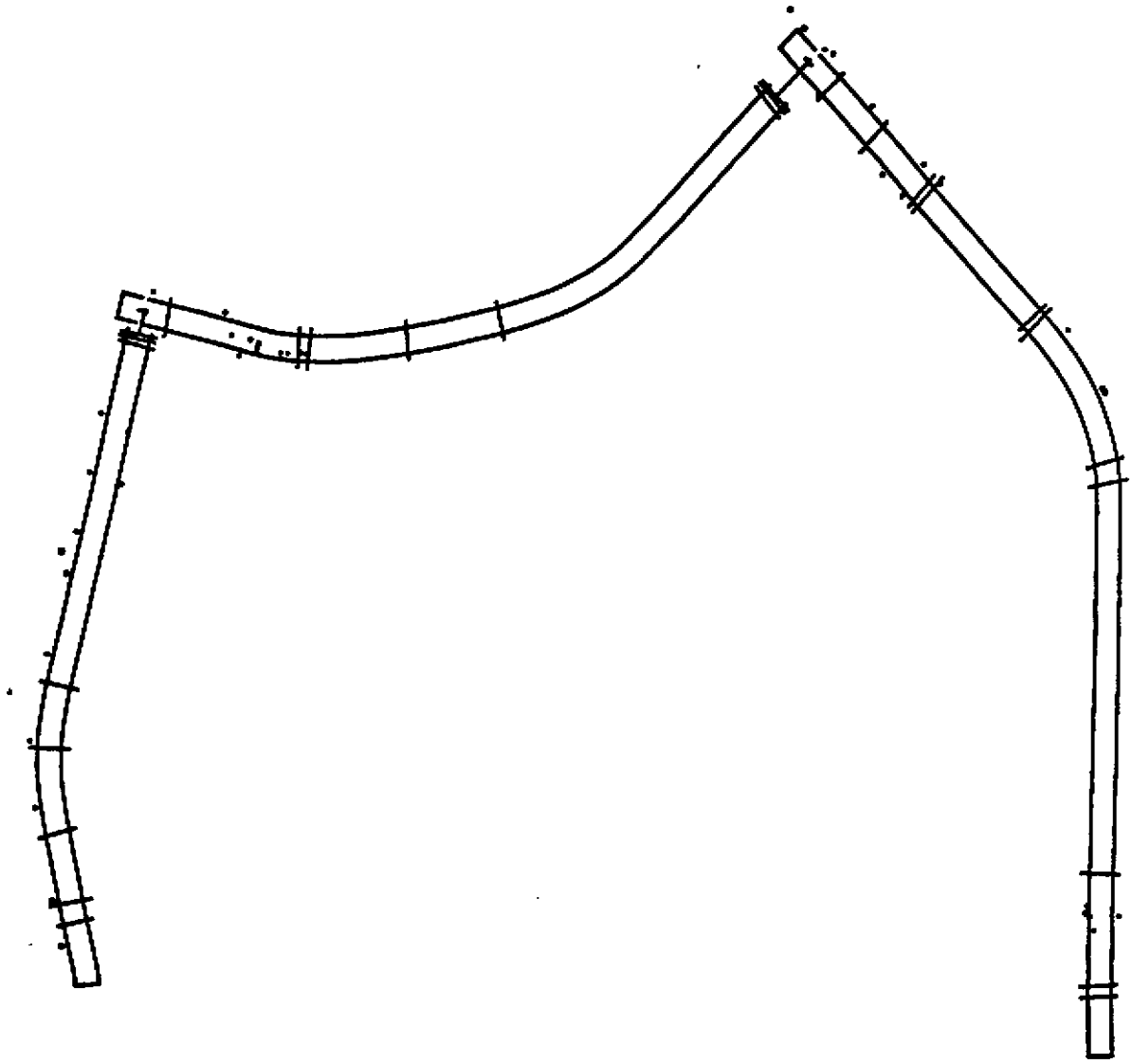


Figure 2.18: Annotated map of a suburban neighborhood, showing roads, intersections, landmark annotations (small circles and dots), and trigger annotations (lines across the road).

2.5.5 Discussion

The main features of EDDIE and the Annotated Maps reflect our current thinking on architectures:

- Task Specific Models: EDDIE does not impose particular connectivity or map structure, but instead provides tools to let users build their own. In particular, much of the data that CODGER put into a central database properly belongs within a single module or pair of communicating processes, as encouraged by EDDIE.
- Explicitness: The most important models maintained by an architecture are vehicle positions. EDDIE explicitly queries the lowest-level hardware for current position, rather than trying to infer vehicle motion from higher levels.
- Architectural Support: EDDIE uses annotated maps as to provide geometric tools at an intermediate level of abstraction, while adding support for soft bumpers, joysticks, and other physical-level control at lower levels. In both CODGER and EDDIE we have left higher, AI-level support to be provided by other modules as needed.

The evolution of our architectures up to EDDIE is a natural one in the evolution of the Navlab project. In the early days, the nature of the modules and their interactions were not known, and our major concern was to not preclude any conceivable system design. Thus, we built CODGER, which was very general and provided easy reconfiguration through anonymity of data storing and access. Now that we know the specific configuration of low-level modules that we need to run the NAVLAB, and how they communicate and synchronize with each other, we seek the simplicity and higher performance that can be achieved by a more specialized architecture. EDDIE is that new design.

2.6 Contributions, Lessons, and Conclusions

We began the Navlab project six years ago with the firm conviction that the best way to make real progress on outdoor mobile robots was to build complete systems, and to concentrate our efforts on eliminating the bottleneck of *inadequate perception*. We continue to agree with, and to follow, those convictions. Following those general guidelines, we have built a number of successful perception, planning, and control modules, and integrated them into systems that drive the Navlab on a wide variety of test sites. During the course of our work, we have also been surprised (usually unpleasantly) by several other aspects of building mobile robots: problems with sensors, difficulty of using experimental computers, questions of how to evaluate our work and how to compare it with results from other groups, and the critical importance of simplicity, and of defining the environment in which the vehicle must operate.

2.6.1 Contributions

Navlab experiments have validated and demonstrated several new ideas.

1. SCARF demonstrates following unstructured roads using color classification. SCARF uses adaptive classification; multiple classes, described by Gaussian distributions in RGB color space; and simple, piece-wise linear road models. These features enable SCARF to follow roads with indistinct edges and changing appearance.
2. YARF uses specialized operators for tracking individual features, combined into a reliable road follower for structured roads. On roads that have lane markings and smooth curves, YARF gains performance by using models of road shape and feature appearance.
3. ALVINN demonstrates neural nets learning to track roads. A single algorithm learns many different roads, with only a few minutes training time for each new road.
4. Our algorithms build accurate descriptions of unstructured terrain from 3-D data. Different levels of descriptions are available, depending on the task requirements and available processing power. This information is directly useful for cross-country navigation.
5. The Navlab builds maps of rugged terrain, combining many noisy 3-D range images to form large-scale maps. Our approach uses a combination of iconic matching, feature matching, and vehicle position sensing. This has been shown before for simple indoor environments, but we invented new techniques and representations for outdoor unstructured terrain.
6. Cross-country trajectory planning requires not only a representation of obstacles, but also reasoning about vehicle capabilities, limits, and inaccuracies. These constraints can be combined efficiently and powerfully, to guide the vehicle up to the limits of their sensing and mechanisms.
7. Simple architectures work best. Dictating the structure of the data and control flow is not needed. It is better to build a toolkit that provides communication, synchronization, map data handling, and clean interfaces to the low-level control, and let individual system builders tailor the system structure to their own needs.

2.6.2 Perception Lessons

Perception: Perception continue to be the bottleneck. That is not to say that the other aspects of mobile robots are solved problems (path planning, map representation, etc.) but rather that they cannot be properly explored until robust perception components are built. The performance of a mobile robot system depends on the performance of the perception components. It is often assumed that robots are control systems, and that perception *will* provide

clean numerical input; or that robots are cognitive problem-solvers, and that perception will provide clean symbolic scene descriptions. Neither of those assumptions are justified by the current state of the art in perception. Robots will not fulfill their potential unless we continue to improve perception capability.

Sensors: While the most important scientific bottlenecks to perception involve inadequate algorithms, the current state of the art of sensor design is also a stumbling block. Too much effort has been spent in overcoming sensor limitations, which is necessary to do real experiments but makes no lasting scientific contribution. A few examples: laser scanning technology is a great advent in 3-D sensing. It still has considerable limitations, however: slow image acquisition which puts a severe limit on the speed of the vehicle, ambiguity intervals, bad behavior on certain material types, etc. Color cameras also have problems: limited field of view, inadequate dynamic range for mixed sun / shadow conditions, unpredictable response from automatic irises and gains, etc. We do not believe that any one magic sensor will "solve" the outdoor robot problem, but advances in sensors will certainly enable and encourage advances in the image understanding algorithms. We continue to build better algorithms, but their full power will not become useful until we have adequate sensors.

2.6.3 Systems Lessons

Design for task and environment: Mobile robots operate in a certain environment to carry out a certain task. In the current state of the art, there is no such thing as a completely general-purpose robot, universal vision system, or generic architecture. Tracking highways requires substantially different processing from driving cross-country. Some of the concepts are shared (local map building, control); and some systems use shared modules, such as neural nets, which adapt to different situations. But currently the right way to build mobile robot systems is to incorporate in the design, from the beginning, knowledge of the task and the environment. Too often, neat ideas are investigated in perception or planning and then artificially matched to an environment and a task. While this is great to demonstrate some new research results, it usually does not contribute much to mobile robots.

Simplicity: The simplest approach is always the best. Designing a complex system does not solve any problems, especially if the components of the system (e.g. perception components) have not even been considered yet. The research community is full of proposed architectural standards that needlessly complicate mobile robots, and that are not based on experience with working perception systems. Simpler is better. For example, the approach that we have followed in our AMV system is to:

1. Define the task: Track roads with the help of a map, and perform actions at specific locations.
2. Develop and analyze the necessary components: road following, object detection, map building.
3. Build and evaluate the components separately to understand their limitations. For example, we first built a smaller system that tracks a road map and stops at specific objects, then expanded to annotated maps and the AMV.
4. Define representations that are matched with the task, such as the annotated maps.
5. Put together components and representations in a system that is configured for the task. The system is "simple" in the sense that it includes only the functionality that is needed for the task using the selected components.
6. Experiment. The important point is that the experimental phase is used to evaluate how well the mission is carried out and to maybe add new perception components, or modify the representations. It is *not* used for debugging a giant complex system.

Computation: Fast computation is of course of great help in building a mobile robot systems. Not only does it improve the performance of the final system, it also holds the promise for more images processed, faster runs, and more experiments, and thus faster progress in the basic research. We have found, however, that faster computation should not be the highest priority. In the early stages of a mobile robot project, especially, the researchers need to try many different possible approaches to perception. It is more important to have easy-to-use computers, with well-supported and efficient compilers, than to have the ultimate in running speed. It is also crucial that I/O be well

supported, both for image digitization and for communicating results. Now, after six years of the project, our algorithms are stable enough that we can properly take advantage of non-standard high-speed machines; but those machines should be stable and well-supported. It is very difficult to do robotics research simultaneously with hardware or operating systems research.

Vehicle: The vehicle itself must be considered an integral part of a mobile robot system, not just a platform on which experiments are conducted. The Navlab was specialized for our early systems, and provides the high-accuracy motion and slow speeds we needed [23]. It was not designed for rough terrain motion, nor for highway speeds. We are currently building new testbed vehicles, that will be capable of the higher speeds that our perception and control can now handle and will be more capable of rough terrain operation. Are testbed vehicles are being selected and modified to complement the capabilities of our sensors, perception algorithms, and planners.

Controller: Real-time mobile robot controllers need to integrate a wide range of capabilities, beyond just control theory: position estimation, mapping and tracking of paths, human interfaces, fast communication, multiple client support, and monitoring vehicle status for safety and debugging. Most mobile robots do not push the limits of current control theory. The major issue in controller design is not control theory, but rather design for system integration.

Debugging and Monitoring: At slow speeds, it is relatively easy to watch the performance of a system. Our first color road trackers, for instance, ran in tens of seconds, which gave ample opportunity for watching graphics, saving the files to disk, noting the response of the vehicle, and so forth. It is much more difficult to debug a system running at higher speeds. YARF now runs in less than a second, which is faster than we can write an image to disk (for later examination), faster than we can examine the debugging graphics, and even too quick to read text output. As a corollary, YARF can now process hundreds of images in a typical run, or thousands of images during a day's experiments, which makes examining the output by hand tedious at best. We need both better technology (faster disks, better video recorders, etc.) and better ideas for debugging complex real-time systems.

Experimental evaluation: Even with proper tools to monitor a particular system, it is difficult to measure progress. The basic problem is how to answer the questions "Does it work?", and, "Does it work better?". Some systems are easy to measure: did an obstacle avoidance system run over a tree or not? Others are more difficult: did the vehicle clip a corner because of bad calibration, bad trajectory planning, bad image processing, or bad control? The problems become worse when comparing work from different research groups. All papers on road following claim success. Most are missing crucial details which would enable evaluating competing algorithms. Even where all the details of the software are spelled out, crucial differences in hardware (processing rates, camera capabilities, vehicle and camera control, etc.) make head to head comparisons difficult. Common image databases provide only a small part of the solution, since different algorithms and vehicles may need different sensor vantage points, image collection frequency, auxiliary data, and so forth.

2.6.4 Conclusion

We are still in the early stages of understanding how to build reliable outdoor mobile robots, both at CMU and in the community as a whole. It is far too early to try to define standards for most modules or architectures. We are still far from being able to design a robot top-down from general specifications, and far from being able to build perception algorithms with specified performance on demand.

The progress in our group and in other groups around the world so far is largely attributable to the experimental approach, and to the emphasis on building complete systems. Mobile robot research is not just research in perception algorithms, or sensors, or architectures, or computers, or vehicles, or controllers. Many fine modules,

developed *in isolation in the laboratory*, have proven difficult to use or incomplete in the context of real outdoor systems. Our greatest advances have come by developing modules to fit a certain system need, using real vehicle data for development and debugging, and testing the modules in the context of a complete vehicle running realistic experiments.

This experimental approach will continue to be fruitful. In the first six years of this project, we have gone from *excruciatingly slow motion (2 cm / sec) in benign conditions* (clean sidewalks) to driving up to the vehicle's top speed (20 mph) on a variety of real roads. There remain big challenges ahead both in driving on roads (handling a variety of lighting conditions, dealing with changing road shapes and lane markings, and handling traffic); and in driving cross-country (moving at higher speeds, mapping terrain and avoiding obstacles). We are working in both those areas. For road tracking, we continue to pursue vision for road tracking, including ALVINN for learning road tracking and YARF for detecting and explaining changes in road shapes. Other projects at CMU are working on strategies for interacting with other traffic, and on tracking moving objects. *We continue to need new sensors, both* for road tracking and for longer-range obstacle detection. Off road, we are working with new range sensors, with inertially stabilized sensor platforms, and with new computer architectures, to build faster and more accurate systems. For both on and off road systems, we are refining our software architecture, continuing the development of maps and planning systems, and building new testbed vehicles.

While general-purpose systems are still far off, the large amount of experimental work over the past few years has brought several mobile robot research groups to the threshold of applications in limited domains. Prototype robots are being proposed or built for several environments. Barren terrain, such as planetary surfaces or some hazardous waste sites, allows easier perception. Limited-access environments, such as underground or strip mines, decrease the need for safety checks and eliminate unknown moving obstacles. Convoy following relies on a person driving the lead vehicle to avoid difficult situations, while subsequent robotic vehicles have the much simpler task of tracking the leader. Other applications involve a human supervising one or more semi-autonomous vehicles, so the vehicles can handle routine cases and decrease operator workload. All these applications will not only be useful in themselves, but will continue to build the components needed for the truly intelligent autonomous vehicles of the future.

2.7 Acknowledgements

Navlab work is the product of many people. Takeo Kanade, William Whittaker, and Steve Shafer have all shared in Principal Investigator responsibilities. Navlab planning and systems have been done by Tony Stentz and Eddie Wyatt. The new controller is the work of Omead Amidi. Martial Hebert is the CMU expert on 3-D perception, including the Navlab's medium resolution mapping. Dave Simon built the first AMV prototype, and Jay Gowdy continues development. Karl Kluge is following structured roads with explicit models, while Jill Crisman and Didier Aubert work on unstructured roads with simple appearance models. Dirk Langer is working on the sonar "soft bumper". Ken Rosenblatt is developing new system integration approaches. Dean Pomerleau, a student of Dave Touretzky, does neural nets on the Navlab.

Thanks also to those who keep the Navlab alive and productive: especially Jim Frazier, Bill Ross, Jim Moody, and Eric Hoffman.

This paper benefited from comments and contributions of figures from many people, especially Dirk Langer, Didier Aubert, Karl Kluge, Omead Amidi, Jill Crisman, Dean Pomerleau, and Jay Gowdy.

This research is sponsored in part by contracts from DARPA (titled "Perception for Outdoor Navigation" and "Development of an Integrated ALV System"), by NASA under contract NAGW-1175, by the National Science Foundation contract DCR-8604199, and by the Digital Equipment Corporation External Research Program.

2.8 References

[1] J. Albus, H. McCain, and R. Lumia.
NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM).
Technical Report Technical Note 1235, National Bureau of Standards, 1987.

[2] O. Amidi.
Integrated Mobile Robot Control.
Technical Report, Robotics Institute, Carnegie Mellon University, 1990.

[3] P. Anandan.
A Computational Framework and an Algorithm for the Measurement of Visual Motion.
IJCV 2(3), 1989.

[4] D. Aubert and C. Thorpe.
Color Image Processing for Navigation: Two Road Trackers.
Technical Report CMU-RI-TR-90-09, Robotics Institute, Carnegie Mellon University, 1990.

[5] P. Besl.
Range Imaging Sensors.
Technical Report GMR-6090, General Motors Research Labs, Warren, MI, 1988.

[6] B. Bhanu, P. Symosek, J. Ming, W. Burger, H. Nasr and J. Kim.
Qualitative Target Motion Detection and Tracking.
In *Proc. Image Understanding Workshop.* Morgan Kaufmann Publishers, 1989.

[7] A. Bobick and R. Bolles.
Representation Space: An Approach to the Integration of Visual Information.
In *Proc. Image Understanding Workshop.* Morgan Kaufmann Publishers, 1989.

[8] R. Brooks.
A Robust Layered Control System for a Mobile Robot.
IEEE Journal of Robotics and Automation RA-2(1), 1986.

[9] R. Brooks.
Solving the Find-Path Problem by Representing Free Space as generalized Cones.
Technical Report A.I. Memo No. 674, MIT , May, 1982.

[10] J. Crisman and C. Thorpe.
Color Vision for Road Following.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 2.

[11] M. Daily, J. Harris and K. Reiser.
Detecting Obstacles in Range Imagery.
In *Proc. Image Understanding Workshop.* Los Angeles, 1987.

[12] M. Daily, J. Harris and K. Reiser.
An Operational Perception System for Cross-Country Navigation.
In *Proc. Image Understanding Workshop.* Cambridge, 1988.

[13] E. Dickmanns and A. Zapp.
A Curvature-Based Scheme for Improving Road Vehicle Guidance by Computer Vision.
In *Proc. 10th IFAC* . Munich, July, 1987.

[14] R. Dunlay and D. Morgenthaler.
Obstacle Detection and Avoidance from Range Data.
In *Proc. SPIE Mobile Robots Conference.* Cambridge, MA, 1986.

[15] T. Dunlay.
Obstacle Avoidance Perception Processing for the Autonomous Land Vehicle.
In *Proc. IEEE Robotics and Automation.* Philadelphia, 1988.

[16] Y. Goto and A. Stentz.
Mobile Robot Navigation: The CMU System.
IEEE Expert , 1987.

[17] Y. Goto, K. Matsuzaki, I. Kweon, and T. Obatake.
CMU Sidewalk Navigation System: A Blackboard-Based Outdoor Navigation System Using Sensor Fusion with Color-Range Images.
In *Proc. First Joint Conference ACM/IEEE.* London, November, 1986.

[18] P. Jacobs and J. Canny.
Planning Smooth Paths for Mobile Robots.
In *Proc. IEEE International Conference on Robotics and Automation.* Cincinnati, February, 1986.

[19] N. Kehtarnavaz and N. Griswold.
Establishing collision-zones under uncertainty.
In *Mobile Robots IV.* SPIE, November, 1989.

[20] D. Keirsey, D. Payton, and J. Rosenblatt.
Autonomous Navigation in Cross Country Terrain.
In *Proc. Image Understanding Workshop.* Morgan Kaufmann Publishers, 1988.

[21] S. Kenue.
Lanelok: Detection of Land Boundaries and Vehicle Tracking Using Image-Processing Techniques. Part I: Hough-Transform, Region-Tracing, and Correlation Algorithms.
In *Mobile Robots IV.* SPIE, November, 1989.

[22] S. Kenue.
Lanelok: Detection of Land Boundaries and Vehicle Tracking Using Image-Processing Techniques. Part II: Template Matching Algorithms.
In *Mobile Robots IV.* SPIE, November, 1989.

[23] K. Dowling, R. Guzikowski, J. Ladd, H. Pangels, S. Singh, and W. Whittaker.
Navlab: An Autonomous Navigation Testbed.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 12.

[24] O. Khatib.
Real-Time Obstacle Avoidance for Manipulators and Mobile Robots.
IJRR 5(1), Spring, 1986.

[25] K. Kluge and C. Thorpe.
Explicit Models for Robot Road Following.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 3.

[26] D. Kuan, G. Phipps, A. Hsueh.
Autonomous Land Vehicle Road Following.
In *Proc. ICCV.* London, June, 1987.

[27] I. Kweon.
Modeling Rugged Terrain By Mobile Robots With Multiple Sensors.
PhD thesis, Carnegie Mellon, July, 1990.

[28] J-P. Laumond.
Finding Collision-Free Smooth Trajectories for a Non-Holonomic Mobile Robot.
In *Proc. IJCAI.* August, 1987.

[29] T. Levitt, D. Lawton, D. Chelberg, and P. Nelson.
Qualitative Navigation.
In *Proc. Image Understanding Workshop.* Morgan Kaufmann Publishers, 1987.

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

[31] T. Lozano-Perez, M. Mason, R. Taylor.
Automatic Synthesis of Fine-Motion Strategies for Robots.
IJRR 3(1), February, 1984.

[32] B. Mysliwetz, and E. Dickmanns.
Distributed Scene Analysis for Autonomous Road Vehicle Guidance.
In *Proc. SPIE Conference on Mobile Robots.* November, 1987.

[33] T. Ozaki, M. Ohzora, and K. Kurahashi.
Image Processing System for Autonomous Vehicle.
In *Mobile Robots IV.* SPIE, November, 1989.

[34] D. Payton.
An Architecture For Reflexive Autonomous Vehicle Control.
In *Proc. of IEEE International Conference on Robotics and Automation.* IEEE, 1986.

[35] D. Pomerleau.
Neural Network Based Autonomous Navigation.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 5.

[36] L. Shaaser and B. Thomas.
Finding Road Lane Boundaries for Vision Guided Vehicle Navigation.
In *Roundtable Discussion on Vision-Based Vehicle Guidance 90.* July, 1990.

[37] S. Shafer, A. Stentz and C. Thorpe.
An Architecture for Sensor Fusion in a Mobile Robot.
Technical Report CMU-RI-TR-86-9, Carnegie-Mellon University, the Robotics Institute, 1986.

[38] R. Smith, M. Self, and P. Cheeseman.
Estimating Uncertain Spatial Relationships in Robotics.
In *Proc. AAAI Workshop on Uncertainty.* 1986.

[39] A. Stentz.
Multi-Resolution Constraint Modeling for Mobile Robot Planning.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 11.

[40] A. Stentz.
The CODGER System for Mobile Robot Navigation.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 9.

[41] M. Turk, D. Morgenthaler, K. Gremban and M. Marra.
VITS--A Vision System for Autonomous Land Vehicle Navigation.
IEEE PAMI , May, 1988.

[42] A. Waxman, J. LeMoigne, L. Davis, and T. Siddalingaiah.
A Visual Navigation System for Autonomous Land Vehicle.
IEEE J. Robotics and Automation RA-3:124-141, April, 1987.

Chapter 3: Annotated Maps for Autonomous Land Vehicles

3.1 Introduction

3.1.1 Motivation

Much of the information that mobile robots need is tied directly to particular objects or locations. Maps, object models, and other data structures store useful information, but do not organize it in efficient and useful ways. We have built a new map-based knowledge representation, the "annotated map", to index information to the relevant object and locations. The annotations are used for a wide variety of purposes: describing objects, providing hints for perception or control, or specifying particular actions to be taken. We have provided a query mechanism to retrieve annotations based on their map locations. We have also built "triggers", which cause a specified message to be delivered to a particular process when the vehicle reaches a given location in the map.

These annotated maps serve a crucial role in enabling missions that are otherwise beyond the reach of autonomous systems. Control descriptors allow mission planners to specify what the vehicle is to do at particular locations, reducing the need for onboard planning. Object descriptors contain detailed instructions of how to recognize a particular object, or contain the appearance of this object as seen by a particular sensor on a previous vehicle run. Such information greatly simplifies the problem of seeing and recognizing objects. Geometric queries enable the vehicle to focus its attention on objects in its vicinity, reducing database access and matching time. The trigger mechanism frees individual modules from having to track vehicle position, allowing them to devote their processing to the task at hand or to lie dormant until they receive their trigger message.

Annotated maps do not by themselves solve difficult problems of sensing, thinking, or control for autonomous vehicles. Their contribution is to provide a framework that makes it easy for other modules to cooperate in planning and executing a mission. Annotated maps thus fill a need that is common to many different vehicles, missions, and architectures.

Many analogous annotated maps exist for human use. Aeronautical navigation charts contain symbolic descriptions of routes (airways) and landmarks, and include annotations such as the Morse code call letters of radio navigation beacons. The AAA produces "Triptiks"[1], which include annotations for route, current conditions ("construction", "speed check"), road type (interstate, two lane, etc.), general conditions ("winds through rolling hills"), points of interest (rest areas, gas, food, and lodging) etc. An intelligent person can usually drive a route without such aids; but they do provide a convenient framework for preplanning, and make "mission execution" easier. Furthermore, as we drive a route, we build our own mental representations of landmark appearance, curves in the road, and so forth, which we use to follow the same route more easily at a later time. Our annotated maps provide the same kind of functionality for autonomous mobile vehicles.

3.1.2 Related Work

At CMU, we have developed a family of autonomous mobile robots over the past ten years. Vehicles have included Neptune, a testbed for stereo vision and path planning [12]; the Terregator, our first outdoor mobile robot [17]; the AMBLER, a walking machine for planetary exploration [2]; and, principally, the CMU Navlab [15, 16]. Our experience, especially with the Navlab, has driven the design of the annotated maps. We already have perception and control modules that can use information from annotated maps, including color vision [5, 9], neural

[1] Triptik is a registered trademark of the American Automobile Association

networks [10], 3-D object recognition [7], and planning [11]. We have also built the EDDIE architecture, which provides inter-module communications, control, and system structure for mobile robots [13, 14]. The tools provided by EDDIE are used for the messages that underly queries and triggers in the annotated map.

Many other groups are working on related problems of mobile robots and knowledge representation. Rather than competing with the ideas of annotated maps, most of this research is providing useful tools and ideas that could use or help generate the annotated maps.

Fennema, Hanson, and Riseman at the University of Massachusetts are building world models and maps for their mobile robot, Harvey [6]. They have defined the concepts of "neighborhoods" (topological regions), "locales" (information to decide whether the robot is within a neighborhood), "milestones" (perception for verification), and actions. The UMass map and plan representations are similar to some of the uses of annotations, but have simple, fixed formats, are focused on declarative representations of 3-D object models, and do not provide map-based triggers.

Rod Brooks at MIT has long argued for simple robots with simple control schemes and simple world maps [3]. We concur that simple, sensor-based maps of particular locations are often useful. The lowest levels of our descriptor annotations are designed to contain precisely the sort of information that Brooks' robots use to calculate their position, or to cause a particular action, in a small local area. We disagree with Brooks' contention that this is the only sort of information that a robot should remember. Robots often work in open, featureless environments, and need precise maps and accurate navigation even where no landmarks may be nearby. Annotated maps are designed to keep precise metric information in the geometric levels of annotations, as well as the lower-level cues advocated by Brooks.

Kender gives a much more abstract view of planning for sensor-based navigation [8]. He describes the combinatorial problem of deciding which sensors to use, and which landmarks should be recognized, in order to reach a given goal. The results of analyses such as Kender's should be entered into triggers, to tell the vehicle what to look for, and into object descriptors, to say how to look for those objects.

Blidberg and his associates at the University of New Hampshire's Marine Systems Engineering Laboratory have implemented world models for underwater mobile robots [4]. Most of their work has concentrated on efficient descriptions of space, such as quadtrees. These spatial descriptions are important, but do not include many of the other forms of knowledge (actions, descriptions) for which annotated maps are useful.

3.2 Scenario

A typical mission for our Navlab mobile robot is a delivery task on unlined, unmodified suburban streets. The Navlab has specialized perception modules, including color vision for road following on major roads [9], dirt roads [5], and suburban streets [10]. It also has 3-D perception, using a scanning laser rangefinder, for landmark recognition and obstacle detection [7]. Inertial navigation on the Navlab is accurate enough to drive blind for short distances [1].

In order to accomplish its mission, the Navlab must use several of these modules. Road following using color vision will follow streets, but will not be able to recognize intersections. Inertial navigation will drive through intersections, but must have an accurate starting position. Landmark recognition will update vehicle position before intersections, but is too slow to be run continuously. Only a combination of all those modules, each running at the appropriate locations, will produce an accurate and efficient mission.

3.2.1 Knowledge and Organization

In general, planning and executing such a mission requires several types of knowledge: what to look for, and how to see it; what to do, and how to accomplish it; where to go, and how to get there. The knowledge may range from high-level symbols, to low-level raw data. Knowledge is both internal to a single module, and used by controlling modules to switch between knowledge sources. Approaching the intersection, for instance, the perceptual knowledge includes:

- symbolic: intersection
- geometric: size and shapes of intersecting roads
- sensor-specific: use laser range finder to pinpoint the position by landmark identification
- raw data: landmark 2 meters tall, 0.4 meters wide at position (x,y)

Control knowledge can also span a range of levels:

- symbolic: turn left at intersection
- geometric: intersection angle 45 degrees
- vehicle-specific: turn with a circular arc of radius 15m
- raw data: steering wheel position left 1200 clicks

This knowledge must be carefully organized if it is to be useful. If the vehicle has to sort through all bits of information it has about every possible object, it will overshoot the intersection long before it has figured out how to recognize it or deduced that it was supposed to turn. It is far better to have information tied directly to the map, or automatically retrieved as needed. The landmark recognition module, for instance, must be able to ask for a description of objects within its field of view, and retrieve the knowledge it needs to recognize them.

3.2.2 Annotated Maps

Annotated maps provide the mechanism for organizing this knowledge, by tying information to a map. The annotations contain knowledge about particular objects, locations, or actions. Annotations come in one of two classes: descriptors and triggers. Descriptors are passive, and are retrieved by queries based on geometry and object type. A query for "all objects of type 'intersection' in this polygon" would return the annotation for the requested intersection, if it were in range. Triggers are active, firing when the vehicle reaches a particular location or crosses a certain line. A trigger will send a message to a particular module, such as "controller: start turning hard left in five more feet".

The knowledge in these annotations comes from many sources, including human experts, mission planning software, and even the vehicle's own observations and experiences on previous missions. It is both declarative (data) and procedural (methods and procedures). The level of the annotations depends partly on the vehicle's computational capabilities. Simple vehicles, in known environments, are able to execute simple pre-planned missions by having every object and action completely annotated at low levels. A more challenging environment, with more variation over time, may require higher-level symbolic descriptors in the map and more reasoning at run time. Practical missions will probably require a mix of levels of detail. Even a sophisticated vehicle may, for instance, decide to record the locations of specular reflections from a mailbox, and use those specularities as recognition cues. It may be much more difficult to reconstruct a 3-D model from the observed data, and to later predict the appearance from the model.

3.2.3 Example Runs

Figures 3.1 and 3.2 show a typical annotated map. Figure 3.1 shows a map of a suburban area, including about 0.7 km of road with two T intersections, and a variety of 3-D objects. Object information was collected using the ERIM laser range finder, and the road information was collected by using the inertial navigation system to provide accurate vehicle positions while we traversed the route. Figure 3.2 shows a detail of the first intersection, including the Navlab's position during a run and several triggers.

The goal of this run was to drive from a house near the beginning of the map to a specified house near the end. Annotations were added to the map to enable the Navlab to carry out this mission. There were annotations to set the speed appropriately: up to 3.0 m/s in straightaways and down to 0.5 m/s in intersections. Other annotations activated and deactivated the module that uses the laser range finder to correct vehicle position based on detected landmarks. Before every intersection there was an annotation that switched driving control from a neural network vision program to a module that used knowledge from the map of the intersection structure and dead reckoning to traverse the intersection. Finally, there was an annotation at the end of the route that caused the vehicle to stop at the appropriate object. The route was successfully traversed autonomously.

In this run, and a variety of other runs, we have successfully used nine different types of trigger annotations:

- set speed
- dead reckon through intersection
- resume vision after intersection
- start landmark matching
- stop landmark matching
- stop at objects
- stop and start fast obstacle detection
- use vision through intersection
- switch perception modules

3.3 Tenets of Map Construction and Use

Several key ideas underly our design for annotated maps, reflecting our experience in building perception and navigation systems for a variety of robots.

Minimize semantic interpretation. No-one can predict all the kinds of knowledge that will be placed in annotations. Moreover, the map module need not understand the annotations. The only common knowledge in annotations should be enough header information to store and retrieve the annotation. All the rest of the annotation belongs to the modules that create it and interpret it, with the format to be decided upon by the module creators. The annotated map serves only as a scratchpad.

No specialized query language is needed. The standard queries ask for all objects of type X within polygon Y. Any query more ambitious than that need not be supported. Any more detailed query would require that the map module know the internal details of each type of annotation. It is more efficient, and a better abstraction, to let the querying module sort through the returned objects.

Separate global position tracking from local servoing. Maintaining the current position estimate in local coordinates is a real-time job, and is best done by the low level real-time controller. In order that locations stored in local coordinates will always be consistent, the controller's local coordinates should never be updated. Commanded trajectories, current positions of obstacles to be avoided, and other phenomena that are used once and then discarded, should be kept in local coordinates and never entered into the map. Map-based calculations, such as matching landmarks against a map, or interpreting a position fix, are aperiodic events best done by a separate Navigator module. The Navigator maintains the transform from local to world coordinates. Any module that needs to know current vehicle position in world coordinates must acquire the Navigator's transform, then apply that to the running position reports of the controller. In practice, acquiring the Navlab's current transform is done in one of two ways, specified at start-up:

- The Navigator can send its transform every time it is updated. This is used by fast-running modules that always need the latest update.
- Slower modules, that have a longer cycle time, may not need every updated transform. Worse, receiving too many updates before the module is ready to read them may cause the input queue to

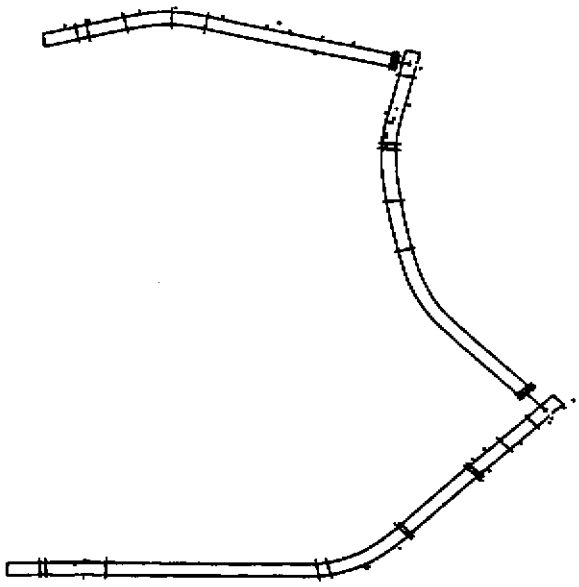


Figure 3.1: Map built of suburban streets and 3-D objects

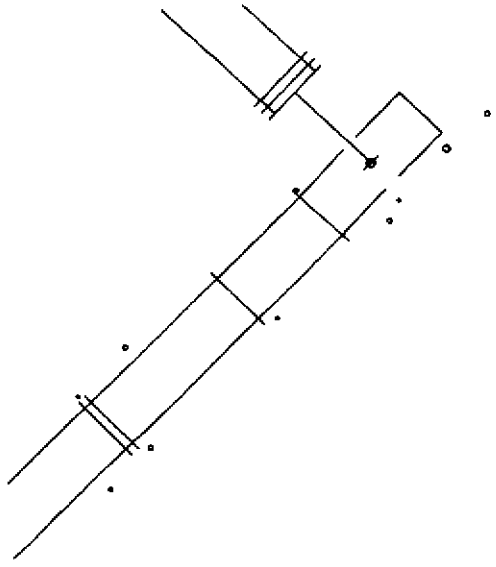


Figure 3.2: Trigger annotations for sensing and vehicle control

overflow. Instead, these modules are notified that a new transform is ready, but do not receive the update until they request it. The Navigator stores which modules have been notified and have not yet requested updates, to avoid sending repeated notifications.

Centralize position tracking. Modules often want to perform specific actions when the vehicle arrives at particular locations in the map. If each module were to continuously poll the Navigator and controller for current position, the controller could become overloaded. Active polling also means that those modules are using computer cycles. Moreover, a Navigator position update may skip the vehicle position estimate past the point for which a module is waiting. For each update, each module would have to figure out if any of its target positions had been passed. We prefer to have a single module, the map manager, doing position tracking for all modules. On reaching the points of interest, it awakens or signals the appropriate module. This is the function of "trigger" annotations.

No master control. The map module is best thought of as an alarm clock (for the triggers) and a scratchpad (for descriptors and trigger messages). It is not some "master" module that controls all thinking, and that therefore can become a major bottleneck. We prefer point-to-point communication between modules, with flow of data and control decided on module by module, rather than forcing all information through a single controller.

Plan incrementally. The map module is designed to be used by many programs, for many purposes, at many times. Some information may be permanent; other annotations may be added to provide directions for only a single mission. It is an advantage to be able to update, add, and delete at various times. In particular, display and user interface modules may read the annotated map from a file, look at it, display the annotations, change things, and write it back out.

3.4 Implementation of Annotations

The annotated map needs to provide efficient access, indexed by position. The annotations themselves need to contain an arbitrary amount of data, with a minimum of externally imposed organization on the contents. We have designed and implemented a two-part representation, consisting of a map grid and an annotation database. Each square of the grid contains a list of any annotations that are included in that square's area.

Adding an annotation to the map is a two-step process. First, the actual annotation is added to the annotation database. Secondly, the map grid must be updated. The location of the annotation is either a point, a line, or a polygon. This location can either be specified directly, for those annotations tied to a location, or retrieved from an object description, for those annotations that describe an object. The location is then scan-converted (converted to a list of cells) into the grid, and a pointer to the appropriate entry in the database is written into each of the corresponding grid cells.

Retrieval of annotations in response to a query is also a two-step process. Queries can specify a polygon and an annotation type. The query polygon is scan-converted into grid cells. The annotations pointed to by each of those cells are collected, checked to see if they match the specified type, and returned.

Triggers work similarly. At each cycle, the map module calculates the current vehicle position. It calculates the line on which the vehicle has moved since the last cycle, and scan converts that line into the grid. Each cell through which the vehicle has moved is checked for trigger annotations. If any are found that have not already been fired, their messages are sent to their destination modules. Since the location of a trigger can be a point, line, or set of lines, a trigger can be fired when the vehicle reaches a certain location or when it enters a given polygon.

3.4.1 Representing Annotations

Annotations are represented with a uniform header format, plus a free-format data field. Typical header fields include:

header:
{type, destination module, used flag, text description, location, next object, previous object, data size}

data:
{pointer to data}

The header portion contains all the information that the map module needs to understand. "Type" and "location" are sufficient for answering queries; "destination" is required for sending trigger messages. The "used" flag is set when a trigger is fired, to avoid firing the same trigger repeatedly if the vehicle stays in the area covered by the trigger for more than one cycle. "Text description" is used by graphics display modules. This information is also sent as part of messages, to make it easier to debug receiving modules. The "location" of the annotation is used both in initially setting up the grid pointers, and for the use of the receiving module. "Next object" and "previous object" are used to describe extended linear objects. Extended objects may also have branches, which meet at intersections. Intersections have a center point, and any number of vertices, each of which points to the beginning of an extended object. The most common extended objects are roads, which are represented as short segments pointing to their preceding or following segments, or pointing to intersections.

The data portion of the annotation is, in the view of the map, an undifferentiated field of bytes. Any internal structure need only be understood by the modules that create and read the annotation. Since the headers have a known, fixed size, they can be stored in a random-access file. The data may be stored as a stream of bytes, with the header containing only a pointer to the beginning of the data and the number of bytes.

3.4.2 Implementation Details

Our prototype implementation has tested some of our design decisions, while other details will be decided after further data collection and analysis.

Grid cell size. If grid cells are too small, queries will have to look at large numbers of cells, and map storage will become a problem. But the querying becomes simpler, because any object found in any of the cells can be returned. Larger cells give faster lookups, but are no longer selective enough to answer queries on their own. Instead, objects within grid cells must still be checked to make sure they are within the query polygon. For autonomous land vehicles with sensor ranges of two to thirty meters, a grid with 0.5 to 1.0 meter cell spacing probably provides the right tradeoff; our current implementation uses 0.5 meter cells.

Handling large maps. For a grid with 1.0 m cells, each square kilometer will contain a million cells. Each cell can be represented with at most a few bytes of data, depending on annotation density. The amount of memory required by a grid this size is easily within the capability of today's computer systems, but for missions spanning several kilometers, we will not be able to keep the whole grid in main memory at once. One possible solution is implementing quad-trees to take advantage of sparse data requirements over most of the grid. A more likely strategy is to keep the grid on secondary storage, and only keep a window around the current vehicle position in main memory. The annotation databases themselves may also need to be kept on backing store, and only read in as needed.

Distributed databases. Object descriptions might be most easily implemented in separate databases, internal to the modules that use them. Then the annotations need only return the index of the database entry. The problem with this method is ensuring consistency between databases in the modules, and indices in the grid. At the opposite

extreme, the map annotations could contain all the data. The disadvantage of this approach is requiring more traffic between maps and objects. An intermediate approach is to start with all the knowledge in the map annotations, but have it automatically replicated in the appropriate modules at system initialization time. This ensures consistency while reducing runtime overhead, at the expense of startup costs. The design of distributed databases interacts with the design for handling large maps. Keeping annotations in individual modules would decrease the amount of information needed by the map module, and thus make building large maps somewhat easier.

In the current implementation, the annotation database is static during a run. When the system is initialized, the user adds stop points, turn points, or other triggers to specify the current mission. When the user is ready, the interface module saves the current annotation database and sends the name of the file to the map module. At start up, each module that needs a copy of the annotation database requests the name of the file from the map module. So modules contain a complete, consistent copy of the annotation database. The map module builds the grid, so it can handle geometric queries. It communicates with the other modules by specifying the index in the annotation database of the objects that match the current query. The map module also watches the grid for triggers.

Map update. Changing an annotation during a run is conceptually easy. Moving objects and annotations is more difficult. If a single object moves, it is easy to erase it from one part of the map and write it into another location. But if an entire portion of the map moves, such as discovering that a portion of the road is really longer than previously thought, the changes can be very hard to handle. Many objects would have to move: the road, all objects attached to it, all landmarks that were seen on previous inaccurate runs and indexed to the road, planned mission steps based on following the road or on seeing those landmarks, etc. It is probably better to note the new information, keep running with the flawed map, and build a new map at the end of this run, rather than try to do updates on the fly. Map update strategy is also influenced by the "large maps" and "distributed databases" design issues. If an individual module updates its copy of an object description annotation, it will need to make sure any permanent information is written out when the run is terminated or when that portion of the map is overwritten by a new data window.

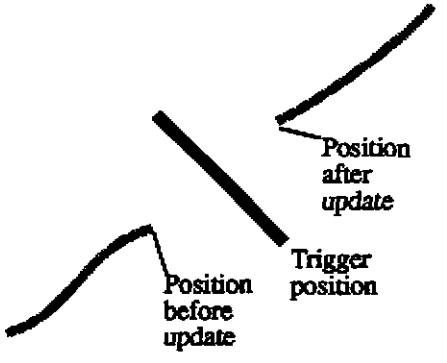
Since in the current implementation, each module keeps its own internal copy of the annotation database, map updates must be specially handled while building a new map. Under most circumstances, the map updates refer to objects that the vehicle will not see again on this run, and therefore the updates need not be propagated to all the modules. At the end of a run, all the new objects can be written to a new map file, to be used on succeeding runs. The exception is for building maps of intersections. Our procedure is to drive through the intersection, following one branch, and building a map; then to reposition the vehicle before the intersection, and follow the second branch. In order to register the two branches correctly, the perception and matching systems need to find newly-mapped landmarks. The map manager writes the annotation database to a file and notifies the relevant modules, which read in the updated database.

Interfaces. Conceptually, it is easy to add annotations to the map. A program reads in the annotation database, adds new annotations, and writes the updated files. Machine-generated annotations, such as object descriptions, use interface routines to read and write the map, and to insert annotations into the annotation database. Annotations added by hand require, besides the basic map interface routines, a user interface to point to locations or objects on the map, type or read the annotation data, display the resulting map, and ask for verification. While the format and contents of the annotations will vary, there is still a large body of common functions that use standard modules. We have built an interface, using X windows, that allows a user to add new objects and triggers to the map. The same interface is also used to display the vehicle and map during a run.

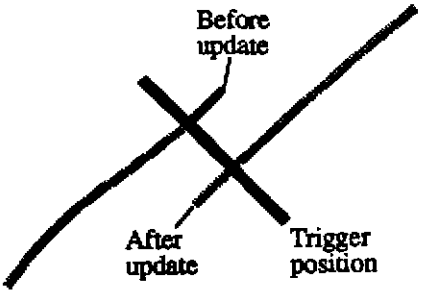
3.4.3 Trigger Details

In order for the map module to track vehicle position, it must know both the controller's current local position estimate, and the navigator's transform that relates local to global coordinates. Position queries to our vehicle controllers are efficient, returning in less than 10 milliseconds. Our current implementation uses an efficient process for getting transforms from the navigator, by having the navigator send the transform each time it is updated. Since landmark sightings or position fixes are relatively infrequent, an event-driven transform update is much more efficient than polling.

When the navigator updates position, the map module has to pay special attention to triggers. It may be that the vehicle position estimate will jump forward, skipping some triggers; or it may be that it will move backwards, creating the potential for firing triggers that have already been fired (see Figure 3.3). If the position update is relatively small, it makes sense to use the line of vehicle travel, plus the "used" flag, to make sure that all appropriate triggers get fired once. If the update is large, it may no longer make sense to fire triggers that should have been fired long ago; and it may make sense to refire triggers that were fired very prematurely. Details of these design decisions are yet to be worked out.



Case 1: position update skips over trigger



Case 2: position update causes retraversal of trigger

Figure 3.3: Problems with refiring mission triggers

The mechanism of notifying a module of a trigger is by sending a message over a port. In the Unix[2] operating system, ports can be set up by broadcasting their address and listening to the net to find out who would like to to talk to them. Once connected, ports appear as files, and can be read and written easily. A module can easily check if there are any bytes waiting on its trigger port. If not, it has not received a message, and can continue running. If so, it can read the message header, allocate the memory structure for the message, and read the appropriate number of bytes into its memory. A running module can periodically check to see if a message is waiting. A sleeping module

[2]Unix is a trademark of Bell Labs

can simply block on read, which will cause it to pause until data arrives. It is possible to set timers, so a module can wait until either a timer expires or a message arrives, whichever occurs first. It is also possible to have an incoming message generate an interrupt, so the module can be notified while running even without checking for incoming data.

3.5 Conclusion

Annotated maps provide a framework to organize knowledge storage and retrieval for autonomous mobile robots. The Navlab group at CMU, and other groups around the world, have many of the individual pieces of a complete system: sensing, sensor understanding, local trajectory planning, control, and vehicles. These pieces in themselves are only sufficient to perform limited tasks. Integrating those components into an efficient system is one of the difficult remaining gaps. The annotated map helps fill that gap. By providing generic data handling, it allows diverse modules to communicate their specialized knowledge. By tying this knowledge to specific locations and objects, the annotated map provides a focus of attention, using an efficient grid structure to answer queries about specific parts of the map. And through the automatic triggers, the annotated map eliminates the need for individual modules to attend to vehicle position and map location. We have built our first prototype annotated map, interfaced several modules to it, and used it to store and retrieve data during real Navlab runs. We are currently addressing the issues of large maps, and continue to interface more modules and to use annotated maps to manage a wider variety of knowledge.

3.6 Acknowledgements

Our work with autonomous mobile robots, and the Navlab in particular, is done with a host of colleagues in the Robotics Institute and School of Computer Science at CMU. Our thanks especially to Takeo Kanade and William Whittaker, co-principal investigators; to Jill Crisman, Martial Hebert, Dean Pomerleau, Didier Aubert, and Karl Kluge, who built the perception modules that the annotated maps support; and to Jim Frazier, who keeps the Navlab running and happy. Martial Hebert, along with our colleagues Stan Dunn, Joe Cuschieri, and K. Ganesan of Florida Atlantic University, provided useful comments on early versions of this report. This research is sponsored in part by contracts from DARPA, titled "Perception for Outdoor Navigation" and "Development of an Integrated ALV System".

3.7 References

[1] Omead Amidi.
Integrated Mobile Robot Control.
Technical Report, Robotics Institute, Carnegie Mellon University, 1990.

[2] J. Bares, M. Hebert, T. Kanade, E. Krotkov, T. Mitchell, R. Simmons and W. Whittaker.
Ambler: An Autonomous Rover for Planetary Exploration.
IEEE Computer , June, 1989.

[3] R. Brooks.
A Robust Layered Control System for a Mobile Robot.
IEEE Journal of Robotics and Automation RA-2(1), 1986.

[4] Steven G. Chappell.
A Simple World Model for an Autonomous Vehicle.
In *Sixth International Symposium on Unmanned Untethered Submersible Technology.* Marine Systems Engineering Laboratory, University of New Hampshire, June, 1989.

[5] Jill D. Crisman and Charles E. Thorpe.
Color Vision for Road Following.
In Charles E. Thorpe (editor), *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 2. Kluwer Academic Publishers, 1990.

[6] Claude Fennema, Allen Hanson and Edward Riseman.
Towards Autonomous Mobile Robot Navigation.
In *DARPA Image Understanding Workshop.* Morgan Kaufmann, May, 1989.

[7] Martial Hebert, InSo Kweon and Takeo Kanade.
3-D Vision Techniques for Autonomous Vehicles.
In Charles E. Thorpe (editor), *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 8. Kluwer Academic Publishers, 1990.

[8] J. R. Kender and A. Leff.
Why Direction-Giving is Hard: The Complexity of Linear Navigation by Landmarks in One-Dimensional Navigation.
IEEE Transations on Systems, Man, and Cybernetics 19(6), November/December, 1989.

[9] K. Kluge and C. Thorpe.
Explicit Models for Robot Road Following.
In Charles E. Thorpe (editor), *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 3. Kluwer Academic Publishers, 1990.

[10] Dean A. Pomerleau.
Neural Network Based Autonomous Navigation.
In Charles E. Thorpe (editor), *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 5. Kluwer Academic Publishers, 1990.

[11] Anthony Stentz.
Multi-Resolution Constraint Modeling for Mobile Robot Planning.
In Charles E. Thorpe (editor), *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 11. Kluwer Academic Publishers, 1990.

[12] Charles E. Thorpe.
FIDO: Vision and Navigation for a Robot Rover.
PhD thesis, Carnegie-Mellon University, December, 1984.

[13] A. Stentz and C. Thorpe.
Against Complex Architectures.
In *6th International Symposium on Unmanned Untethered Submersibles.* June, 1989.

[14] Charles E. Thorpe.
Outdoor Visual Navigation for Autonomous Robots.
In T. Kanade, F. C. A. Groen and L. O. Hertzberger (editor), *IAS-2.* CIP-Gegevens Koninklijke Bibliotheek, Den Haag, the Netherlands, 1989.

[15] Charles E. Thorpe.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990.

[16] C. Thorpe, M. Hebert, T. Kanade and S. Shafer.
Vision and navigation for the Carnegie-Mellon Navlab.
IEEE PAMI 10(3), 1988.

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

Chapter 4: The Warp Machine on NAVLAB

4.1 Introduction

The Carnegie Mellon Warp machine is a systolic array computer developed by H. T. Kung's group, and used for many applications including image processing and mobile robot control [1]. We relate the history of the use of the Warp machine on NAVLAB (**Navigation Laboratory**) and evaluate the Warp machine in light of this experience. As we will demonstrate, the Warp and NAVLAB projects influenced each other in several ways; this influence led to increased capabilities in the Warp machine and useful applications experience, as well as increased capabilities for NAVLAB.

We begin with a short history of the Warp machine on NAVLAB. Next we describe the major NAVLAB systems that were implemented using the Warp machine. Then we evaluate the Warp machine using experience from these systems.

4.2 History of the Warp machine on NAVLAB

This section traces the history of the development of the machine, its software, and its application on NAVLAB, and discusses the motivations that led to key decisions. The earliest systolic array designs that led to the Warp machine were two-level pipelined arrays by Kung, et al. [6, 9], described in the early 1980s. The systolic array formed one pipeline, because the linear array of cells could pipeline data from one cell to the next, and within each cell the floating-point pipeline formed another. These designs were shown to be capable of convolution.

With the introduction of the Weitek 1032 floating point chips in 1983, it became possible to implement a powerful machine based on these ideas using ordinary engineering effort—i.e., without custom VLSI and with a moderate number of processors. A machine using these chips was designed in the fall of 1983, and it was shown to be capable of performing one- and two-dimensional convolution as well as the Fast Fourier Transform (FFT) [7].

At this point the Warp cell included the Weitek floating point chips, which were fed data from a pipelined register file, two input and output queues connecting each cell, and some onboard cell memory [8]. Addresses were supplied externally via a third queue. No data-dependent branching or address generation was possible.

In early 1984 a group of researchers, including hardware, software, and applications designers, began planning the design of the Warp computer. The design of the machine changed rapidly, and became much more general. Data dependent control flow and program memory was added. A crossbar, originally with limited interconnection and later with full generality, was added to connect the various functional units on the cell. The pipelined register files were replaced by random access register files.

The cell at this point had several features that were eliminated later. The address queue was still the sole source of addresses for cells. It was thought that address generation for complex addressing operations such as FFT could be factored out from the array and performed on a special board called the Interface Unit. The address and data paths between cells fed into RAMs with read and write counters that could be incremented or held—not queues. This made it possible for the queue to be used as an auxiliary scratchpad register file. However, it was not possible to switch back and forth between using the RAM as a register file and as a queue, since the counters could not be saved or restored. Thus, using the RAM as a scratchpad register file eliminated one data path to the cell. There was also a "loopback" feature where the output of a cell could be fed back into its input queue in order to allow a cell to simulate multiple cells.

The Warp cell was built in prototype form, as a two-cell array with an interface unit, called the "demonstration"

machine. Assembler development proceeded in parallel through two stages: first at the 100 ns, 16-bit word level (summer of 1984), and later at the 200 ns, 32-bit word level (fall of 1984). Even before assemblers or simulators were available, programs such as affine transformation, clipping, histogram, median filtering, and binary image processing were being written.

Carnegie Mellon selected General Electric and Honeywell as industrial partners to help in the later design and build the full-scale machines at the beginning of 1985. They participated in the design and construction of the first two-cell demonstration machine and in the design of the external host software.

The first demonstration Warp cell array was completed and demonstrated in mid-1985. The array consisted of two Warp cells and an interface unit. The array was controlled and fed data by a Sun 2, which also ran applications code not running on the Warp array. (The external host had not yet been completed).

FIDO, a stereo vision system used to drive a robot vehicle, was a key application of the Warp machine in the early part of the project. It was proposed to speed up FIDO by a factor of ten, from about 30 seconds/step to about 3 seconds/step. Implementation of FIDO algorithms on the Warp machine started in the summer of 1984.

The start of the Parallel Vision and Road Following projects in January of 1985 led to early use of the Warp machine in real situations. In most hardware projects, applications of the hardware to real problems occurs only after much of the software and hardware is already developed. But these projects helped provide focus and direction for the Warp project even as the hardware and software were being defined. A simple color-based road-following program was implemented on Warp in July, 1985, and used to drive the Terregator in the fall. To our knowledge, this is the first application of a supercomputer to actual control of a robot vehicle. These runs set records for speed and distance (up to several hundred meters at 0.5 km/hr) of the Terregator.

In parallel with the applications of the demonstration Warp array, the development of the W2 compiler proceeded. Early in the compiler's design, it was realized that a design error in the cell made it difficult to generate code efficiently. The problem was that on transfer of a word of data from one cell to another, the receiving cell had to explicitly increment its queue counter when the word arrived [1, Section IVa]. In order to generate such code, the sending and receiving loops had to be unwound three times in general. This led to very large code bodies. A hardware change was completed by the end of September, at which time the old machine (and W1 programming) was retired and the new machine (with W2) was used exclusively.

In December of 1985 we began serious plans for installation of a Warp machine on NAVLAB. Since NAVLAB was to be a self-contained machine, it was not practical to do the image processing remotely using the Warp machine, as we had with Terregator. But installing a new computer like the Warp machine in a moving environment required careful planning, both to ensure that it was useful to NAVLAB and to guarantee that this almost-unique machine was not damaged. Issues like cooling and vibration of the Warp cell array were considered in particular. In fact, as we later learned, the critical issues were cooling of the external host MC68020 processor and memory boards and connector damage as the Warp cells were removed and replaced in the backplane. Cooling of the Warp cells was not difficult because they dissipated much less heat per area than the commercial external host boards, which were tightly packed with chips. Vibration was easily dealt with by ordinary measures like mounting a plate to hold the Warp cells in place, and mounting the rack holding the Warp machine with shock-absorbing mounts.

As the first full-scale prototypes were being built, we began to look forward to the production Warp machines, which would be built using printed-circuit boards. The change from wirewrap to printed-circuit boards allowed some redesign; in particular, we reimplemented the cell input queues with special-purpose chips, and eliminated the loopback feature, freeing a lot of board area. This area was used to expand the cell data and program memory by a

factor of eight, add another register file and local address generation, and add local control so a cell could be blocked if it tried to write to a full queue at an adjacent cell, or read from an empty queue. The result of all these changes was to create a much more powerful cell, with flexibility comparable to a standard computer. The extensive changes (particularly the blocking mechanism) required considerable redesign time. The first full-scale PC Warp machine (called the "production" machine) was accepted at Carnegie Mellon from General Electric in the spring of 1987.

Towards the end of 1986 Hamey began developing the Apply compiler. Apply had been previously developed as a C subroutine package for writing image processing functions; the programmer would write a simple subroutine that processed a window of an image and the Apply subroutine would "apply" the subroutine all across an image. This was done to speed up image processing using a subroutine package for accessing images in different formats; the C Apply subroutine buffered the image especially efficiently. Hamey adapted the C subroutine idea to a code generator for the Warp machine that took W2-like Apply programs and generated W2 programs. In the summer of 1987, Wu developed the first "full" Apply compiler, that took Ada-like Apply programs and generated W2 code [5]. This compiler took advantage of the Warp cell's capabilities and generated efficient W2 programs for local image processing functions. At the same time, Ribas developed a library of approximately one hundred Apply programs.

The new Warp prototype was used extensively with Terregator in stereo vision, obstacle avoidance (using the ERIM scanning laser rangefinder), as well as color-based road following, from April to August 1986.

The NAVLAB work on the Warp machine began in June 1986. The work included development of a geometry module for color road classification, which was tested from the beginning on the Warp machine. One of the wirewrap prototype machines was mounted in NAVLAB January 1987, and demonstrated color-based road following and ERIM-based collision avoidance in the spring of 1987. Both the color and the ERIM code were run on the same Warp machine; we thought we could get better performance with two Warp machines, and tried this idea later in the year.

In the course of integrating the Warp machine into NAVLAB we replaced a complex linking procedure that combined the C program calling the W2 Warp program with a runtime code downloading interface. The same interface supported remote procedure call of Warp routines over the Ethernet. As a result, in early 1987 we constructed a runtime code downloading procedure together with an interface that allowed calling Warp routines remotely with an Ethernet interface. This interface greatly aided development of Warp code.

A second, smaller (four-cell) PC Warp system was mounted on NAVLAB in November 1987. With two separate Warp arrays we could do the ERIM processing in parallel with the color-based road following. This two Warp machine system was demonstrated at the end of 1987. However, there were serious problems with mounting a second Warp machine on NAVLAB. The Warp cells were not a problem; a ten cell array could easily be split into two arrays. But the external host boards had to be duplicated, something which we were loath to do because of the expense. Moreover, the external host was one of the least reliable components; duplicating it reduced its reliability correspondingly.

In this period we seriously addressed the issue of cooling for the Warp machine, particularly its external host. We installed a special air conditioner for Warp, and added temperature sensors that would automatically turn off the Warp machine when the temperature went too high. This allowed us to run the Warp machine continuously on NAVLAB, giving us a three Warp system; two in the laboratory, and one on NAVLAB, which could be used remotely when the NAVLAB was at home and connected to Ethernet.

The major application of the Warp machine from here on was color-based road following. In the spring of 1988 an adaptive color classification algorithm, using one or two cameras (one with the iris wide open and one with the iris nearly closed, to increase the dynamic range) and a simple geometric model was implemented. The SCARF road following algorithm was re-implemented on the Warp machine in October with a speed of four seconds per image. This was sped up to two seconds per image in November. The resulting system was demonstrated in December; NAVLAB was driven at one meter/second, with a processing speed of ten to one hundred over the same algorithm running on the Sun. SCARF speed was further improved in February 1989.

At this point development of ALVINN, for neural net-based road following, began. A three-layer neural net was trained to recognize driving direction in graphics-generated road images. The training was done off-line, in an eight-hour run on the Warp machine in the laboratory. The resulting trained network was then used to drive NAVLAB. Runs began approximately in February of 1989. Images could be processed as quickly at 0.75 s/image; on March 16 a new NAVLAB record of 1.3 m/s was set. Later, in June, we found that we could train the network "on the fly" by feeding it live road images and driver steering angle while NAVLAB was under human control. This training was done using the Warp machine on the NAVLAB. The resulting technique was very powerful; we could for example train the network by driving the NAVLAB halfway along a test course under driver control, and then allow the network to take over vehicle control.

In order to see further speed improvement both in the color-based and the neural net-based road following work it was thought that the Sun 3 Warp host should be replaced with the newly available Sun 4. To do this for the Warp host would involve extensive changes to the Warp software. Moreover, the Sun 4 was several times more powerful than the Sun 3 and its integrated, general-purpose nature made this power more usable than the Sun 3/Warp machine combination. The Sun 4 also required less power, space and cooling, which were critical limitations on NAVLAB. Accordingly, the Warp machine was taken off NAVLAB September 6, 1989, and replaced by a Sun 4.

4.3 FIDO

FIDO (Find Instead of Destroy Objects) was a stereo vision navigation system used for the control of robot vehicles; it included a stereo vision module, a path planner, and a motion generator. This system descended from work done by Moravec at Stanford [12]. After Moravec came to Carnegie Mellon in 1980, work was done by Thorpe and Matthies [11, 14], who gave the system its name. More recently, work was continued by Klinker, Crisman, and Clune [2] as well as others. This vision system was unusual in its longevity and in the range of speed over its span of development: Moravec's original algorithm, which was heavily optimized (though different in many important ways from the FIDO algorithm), took fifteen minutes to make a single step while running on an unloaded DEC KL10; the Vax 780 implementation ran at thirty-five seconds per step; the Sun 3 implementation took 8.5 seconds per step; and the implementation on the Warp machine took 4.8 seconds per step.

4.3.1 FIDO Algorithm

FIDO was a feature-based algorithm. A *feature* was a point that was detected with FIDO's interest operator and located in three-dimensional space by correlation between the left and right images. An *obstacle* was a feature that the vehicle could not drive over – i.e., a feature sufficiently above ground level. It was assumed that all actual obstacles to the vehicle would have enough image features to be detected by FIDO as obstacles.

FIDO performed the following steps, as shown in figure 4.1. First, it took two 512×512 images of its environment, a left image and a right image. These two input images were reduced by the *Image Pyramid Generator* by successive factors of two, creating images of size 256×256, 128×128, and so on. Then *Image Pyramid Correlation* was used to locate all of the previously known features in the new right image. If the feature

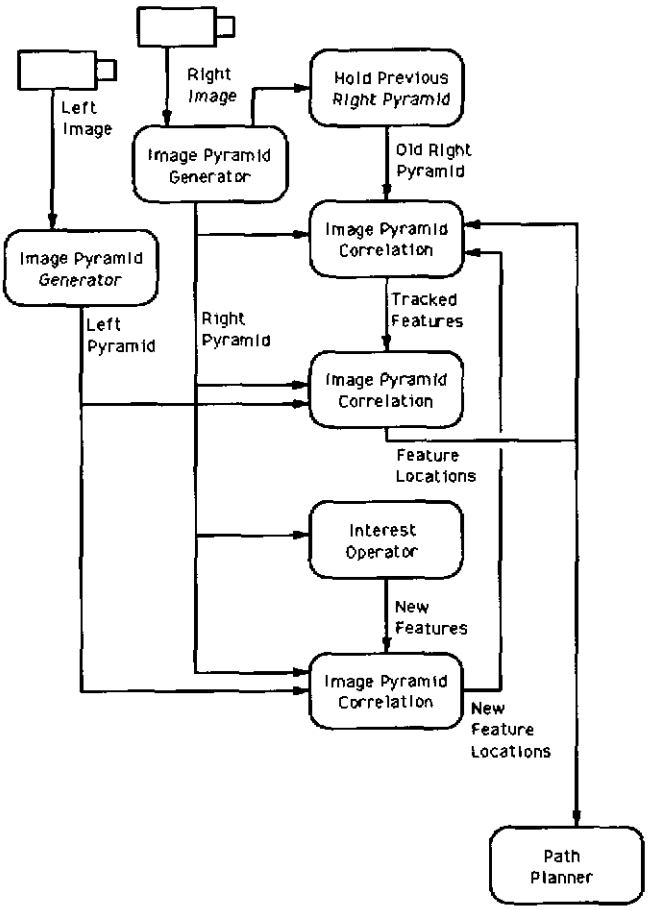


Figure 4.1: FIDO Block Diagram

could be seen in the new scene, it became a tracked feature. *Image Pyramid Correlation* was then used to identify the tracked feature from the new right pyramid in the new left pyramid. Once the corresponding features were located, the three-dimensional position of the obstacle creating that feature was identified and the vehicle was directed one step towards its goal by the *Path Planner*. The *Interest Operator* picked new features in the right pyramid so that new obstacles moving into the scene could be detected. Again *Image Pyramid Correlation* was used to find the corresponding point features in the left image pyramid. The new features and the tracked features were combined to form the new list of "previously known" features for the next image.

4.3.2 Implementation of FIDO on Warp

FIDO has been implemented on the demonstration Warp system as well as the prototype Warp machine. In the summer of 1984, Dew, Chang, Matthies and Thorpe designed a new version of the FIDO system to run on Warp, which was then in its initial design phase. They identified the three major vision algorithms (correlation, interest operator, and pyramid generation), which were considered to be suitable for implementation on a systolic array such as the Warp machine. Then they redesigned FIDO to run on the Warp machine. Next the three vision modules were implemented using Warp microcode by Klinker on the demonstration Warp system. Later, when the prototype Warp machine was available, the modules were reimplemented by Clune using W2 and the external host and the Warp array ran parts of the algorithm in parallel.

Each of the modules that were implemented on the Warp machine will be now described and their performance

will be given.

4.3.2.1 Image Pyramid Generation

The image pyramid consisted of seven levels, starting with a 512×512 image and ending with an 8×8 image. Areas of 2×2 pixels were replaced by one pixel in the next level of the pyramid. The new pixel value in a lower resolution image was computed by averaging over a window in the higher resolution image. The simplest averaging was to take a 2×2 pixel area and average it to one pixel. The initial implementation on the Warp array used overlapping 4×4 windows, which gave slightly better results than 2×2 windows.

The pyramid generation algorithm was implemented in W1 in a systolic scheme, as suggested by Kung for convolution-type algorithms [7]. The algorithm accumulated sixteen pixels in a 4×4 window and then normalized to produce one reduced pixel value. This was mapped onto the Warp array as nine modules, with the first eight each adding two new pixel values to the accumulated partial sum, and the ninth module normalizing the result. The second, fourth and sixth module also stored the partial results until the necessary pixels from the next row underlying the 4×4 window had arrived at the module. The new data and the partial results were then sent together to the next module.

A simpler sequential algorithm (with non-overlapping reduction windows) took about one second on a Vax/780. Nine Warp cells provided a speed-up of 14, which was relatively small. The implementation of the pyramid generation algorithm was communication intensive: it used the adder effectively only half of the time (in every other row). It did not use the multipliers at all (except for a normalization). Each Warp cell was used as a 2.5 MFLOPS machine, for 25 MFLOPS from the array. This explains the relatively small speed-up of the pyramid generation algorithm. Adding more cells would not increase the speed since this would not reduce the communication requirement.

This module was later reimplemented as a C program to run on the cluster processors, since very little computation was done here. This made it possible to do the two pyramid generations in parallel using the two cluster processors. In this implementation non-overlapping 2×2 windows were used instead of the overlapping 4×4 windows in the implementation on the Warp machine, to simplify the computation.

4.3.2.2 Interest Operator

FIDO detected features with an interest operator, which was designed to detect points that could be localized well in different images (for example corners). Such points had image intensities that changed rapidly in all directions. The interest operator took squared pixel differences in the 3×3 neighborhood around the point [3, 14]. The output of the operator was the minimum of the squared differences in the vertical, horizontal, and both diagonal directions. The interest values were locally maximized in one hundred subimages that were arranged in a 10×10 grid. The maxima of all subimages were stored in a list ordered by decreasing interest values. This gave a set of point features, distributed across the image, which could be localized in other images.

Only the first part of this algorithm, accumulating squared pixel differences in all four directions for every pixel, was implemented in W1 on the Warp array. In the demonstration system, the processing stopped here. Later, we implemented the minimization, maximization, and list formation on the cluster output processor.

The interest operator did not offer a good partitioning into modules with similar timings. We thus did not try to implement it in a systolic scheme, as was described for the pyramid generation algorithm in the previous section. Instead, we used the *input partitioning* model [10] where data was divided into equally sized parts. In this scheme, each cell performed the complete algorithm on a portion of the data. An $m \times n$ image was divided into c vertical stripes to be processed on c different cells. For the interest operator, the stripes had to overlap by four pixels, due to

the width of the operator window. Thus, every cell ran on $m \cdot (\lceil \frac{n}{c} \rceil + 4)$ pixels. The systolic communication facilities were then used like a "bus": each cell received data from the previous cell and sent it to the next cell. The host sent the data interleaved such that each cell could use every c^{th} pixel for itself. At the beginning of every new iteration, c new pixels were sent over the "bus." The offset between programs that ran on neighboring cells was two cycles such that each cell started a new iteration exactly when a new pixel arrived.

The sequential algorithm ran in about 2.65 seconds on a Vax/780. Ten Warp cells provided a speed-up of 26.5. The adder was the most used resource of the interest operator. It was used in forty out of sixty-five cycles of the innermost loop. The multiplier was barely used (four multiplications in sixty-five cycles). The algorithm thus used each cell as a 3.4 MFLOP machine. The addition of more cells would greatly improve the speed. In the described implementation, each cell needed a new pixel every sixty-five cycles. Thus, maximally sixty-five cells could have been used in parallel before the interest operator had become I/O limited.

4.3.2.3 Image Pyramid Correlation

For a given pair of images and a given list of point features in one image, the correlation algorithm found the corresponding point features in the other image. The search for the most likely correspondence was performed on the image pyramids, starting at the lowest resolution (8×8) image. At each level, a 4×4 template around the interesting point was correlated with an 8×8 search area in the other image at the same resolution. The best matching position of the template in the search area determined the position of the search area in the next higher resolution image in the pyramid [12].

A pseudo-normalized correlation was used, as given by this formula [3]:

$$CORR_{lm} = \frac{S_t - t_{mean} S_1}{t_{var} + (S_2 - S_1^2)/16}$$

$$\text{with } S_1 = \sum_{i,j=0}^{3} I_{i+l,j+m}, \quad S_2 = \sum_{i,j=0}^{3} (I_{i+l,j+m})^2, \quad S_t = \sum_{i,j=0}^{3} t_{ij} I_{i+l,j+m}$$

where t_{ij} denotes the template element at position (i,j), and $I_{i+l,j+m}$ denotes pixel at position $(i+l,j+m)$ in the image.

In the W1 version of the algorithm, the Warp machine found the positions of all features for one given pyramid level at a time. First, templates for all pyramid levels were sent. The cells stored the templates and computed their means and variances. Then the search areas of each level were given to the Warp array in the same sequence as the templates. The cells correlated the current template with the current search area and sent the correlation results for every template position to the output cluster. The cluster processor then found the best position of each template within its search window and determined the search areas for the next higher resolution. The process was repeated for all of the images in the pyramid [3].

The correlation algorithm was implemented in a systolic programming scheme, just as in the pyramid generation algorithm. It was designed as nine modules. Each of the first eight modules covered two template elements. The algorithm was designed so that initially, each module received the template elements and stored the respective template elements of each template. The mean and the variance of all templates were computed and stored in the ninth module. Then, in the correlation phase, each module got the pixels of the search areas and the partial sums S_1, S_2, and S_t from its left neighbor and updated the partial sums before it sent them to its right neighbor with the next pair of pixels. As in the case of data pyramid generation, the second, fourth and sixth module stored the derived partial results until the pixels of the next row, underlying the current window position, arrived. The ninth module

combined the partial sums and the mean and variance of the current template into a correlation value that denoted how well the template fitted the data in the search area at the current position.

The sequential algorithm took about 2.3 seconds on a Vax 780. Nine cells provided a speed-up factor of seventy-eight. This was a much higher speed-up than that achieved by the pyramid generation algorithm and the interest operator because the multiplier was used in every cycle and the adder was used in every other cycle. Each cell thus ran here as 7.5 MFLOP a machine. The communication facilities were also used in every other cycle. Therefore, the correlation algorithm was a fairly well balanced algorithm. The maximum speed-up would have been reached if eighteen cells had been used (due to communication requirements).

This module was originally written as a systolic program, but could not be reimplemented in W2 in this way because the prototype W2 compiler allowed only homogeneous code. Instead, it was implemented using input partitioning, like the interest operator.

4.3.3 Performance of the Vision Modules

The reimplementation of FIDO led to a total system time for one step of 4.8 seconds, which was a large speedup over the original time, but still relatively small compared to the time that had been achieved by that time on a Sun 3 alone (8.5 seconds). In this section we will analyze the performance of the FIDO system on the Warp machine.

Most interesting was the pyramid generation module on the Warp machine. It actually took longer to run on the Warp machine than on the Sun alone. This was because the data flow between the clusters and the Warp machine was unbalanced. Time consuming manipulations were required to order the data correctly for the Warp machine in this implementation, but the actual pyramid generation on the Warp array was not computationally intensive. The array was virtually starved for data. This was a case where the ordering of data was too complex for the Warp machine (specifically the clusters). A more efficient implementation would be for the cluster processors to send the pixels in the order that they were stored in memory so that data could flow rapidly into the array, and the Warp array could reorder the data.

The interest operator and correlation functions did not perform at the predicted speeds on the prototype machine, although they were faster than the comparable Sun functions. If the startup times on the Warp machine were subtracted (the startup time was much lower on the production Warp machine), then the actual times were close to the predicted times.

The interest operator required about 0.1 seconds of Warp array processing time for the ten cell implementation compared with a one second Sun 3 time. Additional time was spent starting the Warp array (about 25 milliseconds). However, most of the time was spent in post processing. After the interest operator was run, the cluster processors sorted and selected the resulting data. This was about 28% slower than the Sun 3 processor, because of a slower clock rate.

The correlation function had less than a factor of three speedup, compared to a Sun 3 alone. As with the interest operator, the time required for the correlation function on the Warp array was small. However the time spent processing data for the Warp array on the cluster processors dominated the total execution time. This time included the following:

- Startup overhead of 25 ms. In one step, correlation was called seven times, for a total overhead of approximately 0.2 seconds.
- Rearranging data for the Warp machine. Complex addressing was needed to send the image patches from the different pyramid levels to the Warp machine.
- Fixed loop function of the W2 compiler. A fixed number of features must be processed in every correlation, in our case fifty, although the average number of features in a correlation was

approximately twenty-five.

Work on FIDO stopped in 1987. The move from Terregator to NAVLAB, with its ERIM laser range scanner, ended it. FIDO's stereo vision was not as reliable, and could not be made significantly faster, than the ERIM scanner. While FIDO could locate a small number of "feature points" in a few seconds of Warp machine time, the ERIM scanner provides a dense three-dimensional array of points in one-half second scanning time and a few seconds of processing. Moreover, ERIM worked much more reliably than FIDO – it could even be used at night, and FIDO's interest operator, designed to look for object corners in indoor images, never performed very well outdoors; it was confused by image clutter, such as leaves, in outdoor images.

4.4 SCARF

SCARF (Supervised Classification Applied to Road-Following) is a road-following navigation system used to drive the Navlab. The system labels every pixel in an image as road or off-road depending on how well the color of the pixel matches road and off-road colors from previous images. The road location is determined by matching an ideal road shape model with the labeled image data. This location is then used to update the stored road and off-road colors and to steer the robot vehicle. This system has evolved over a period of four years and is still being used as a research tool today.

SCARF has had several implementations on the Warp machine. The first of these implementations was written in W2 and Apply on the prototype Warp machine. This implementation showed only a factor of two speedup over the Sun 3 version of the code. Later versions of SCARF were implemented on the production machine. We used the Warp machine to process two larger images rather than the one smaller image of the prototype implementation. In this case we saw a speedup of six over the Sun 3 implementation. The fastest SCARF system had a total one second Warp machine time and a total time of three seconds counting all the overhead including vehicle control. Compared to a Sun 3 implementation, the speedup was thirty for the Warp machine time or ten counting all overheads.

In the next section, we will describe the SCARF algorithm in more detail. Then we will show equations for each SCARF module that was implemented on the Warp machine and describe their implementation. Finally, we will discuss how the later SCARF implementations were derived from the first and discuss in general terms the timing of the systems.

4.4.1 SCARF Algorithm

The program flow and data transfer between the different SCARF modules are shown in Figure 4.2. SCARF starts with (480×512) RGB images from the color camera. The *Image Pyramid Generator* creates an image pyramid for each of the RGB input images. The *Texture Operator* takes the blue image pyramid and creates an image corresponding to the texture seen in the scene. The texture image and the smallest level of the RGB pyramid, the *RGB Images*, are sent to the *Classifier*. The *Classifier* compares the color of each pixel in the image with remembered road and off-road color described by the *Color Model*. Each color pixel is assigned a value in the *Probability Image* representing the likelihood that the pixel is a road pixel. This image is used as voting weights by the *Road Hough* module. Each pixel votes, using its assigned probability, for all of the possible roads that contain that pixel. The *Road Location* with the largest accumulated vote is selected as the best road. The resulting *Road Location* is used by the *Color Model Generator* to label pixels in the image as road and off-road. The labeled pixels are then used to formulate new road and off-road colors models. The *Road Location* is also used to generate motion commands for the vehicle.

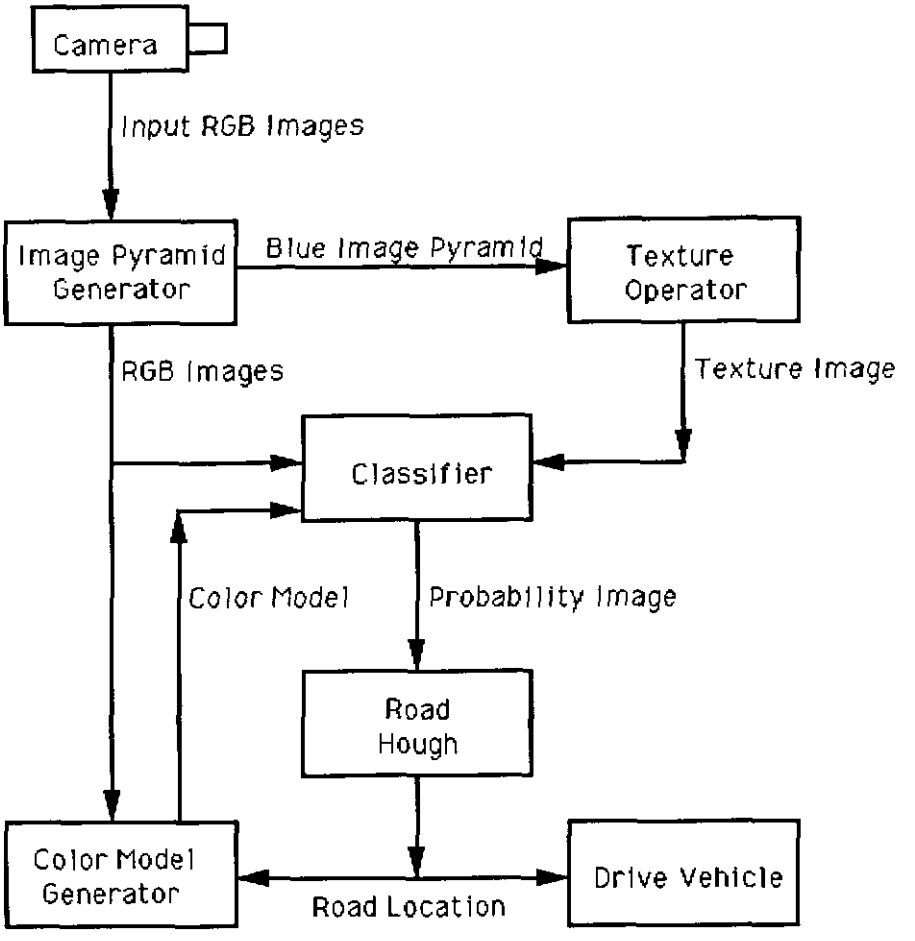


Figure 4.2: SCARF Block Diagram

4.4.2 Implementation of SCARF on the Warp machine

SCARF has been implemented on the prototype Warp system as well as the production machine. In the initial system, four modules were picked for implementation on the Warp machine: the *Texture Operator*, the *Classifier*, the *Road Hough*, and the *Color Model Generator*. The modules were initially implemented by Crisman and Webb for the prototype machine. Later, Chen and Crisman implemented a different version of SCARF on the production machine. This version was more computationally expensive than the original system. A final SCARF system was implemented by Crisman. It had only one module to process the entire SCARF algorithm.

The next sections will describe each module that was implemented in more detail. The general equations will be given and a brief overview of how the algorithms were divided among the Warp cells. Finally we will discuss how these modules were combined to form the SCARF system and give overviews of their performance.

4.4.2.1 Texture Operator

The *Texture Operator* consists of two Roberts' edge operators and a *Texture Determination* operator. The first Roberts' operator is run over the 120×128 blue image to form a 120×128 *Fine Edge Image*. The second Roberts' operator is run over the 30×32 image to form the 30×32 *Coarse Edge Image*. A Roberts operator computes an edge value by looking at the input image, *in*, values around the edge pixel location. Therefore edge value at row i and column j is calculated by

$$edge[i][j] = |in[i][j] - in[i+1][j+1]| + |in[i+1][j] - in[i][j+1]|.$$

The final phase of the *Texture Operator* is the *Texture Determination* operator. It first creates a *Fine Texture Image* and then counts the fine texture pixels in a region to form the smaller, output *Texture Image*. The *Fine Texture Image* is computed from the *Fine Edge Image*, the *Coarse Edge Image*, and the *Average Image*. The *Average Image* is the 60×64 input blue image. The pixel located at row i and column j of *Fine Texture Image* is calculated by

$$fine\_texture[i][j] = THRESHOLD\ (fine\_edge[i][j] / (\alpha\ coarse\_edge[i/4][j/4] + (1-\alpha)\ average[i/2][j/2]\ )).$$

THRESHOLD is a thresholding function that outputs a 1 if its argument is greater than a particular threshold value and 0 otherwise. The constant α=0.2 is a weighting value; it was set heuristically. The purpose of the *Fine Texture Image* was to locate texture in the input image that was independent of brightness and scale.

The implementation on the Warp machine used three different modules, the first two of which were Roberts edge operators and the third was a combination of the *Texture Determination* operator. Although the algorithm was the same for the edge operators, they needed to be implemented separately since the input images were different sizes.

The edge operators were written in Apply and the last module was implemented in W2. The input images were divided column-wise among the cells. To speed up the processing time, the loops were unwound and each pixel of the output *Texture Image* was calculated immediately after the calculation of the corresponding 4×4 block of the *Fine Texture Image*. Therefore there were 16 explicit equations in the W2 code for each of the fine texture pixels in the block, each of which was followed by a counter keeping track of the sum.

4.4.2.2 Classifier

The classification module of SCARF uses a Bayesian classification technique to determine the likelihood that each pixel is a road pixel by matching pixel colors with remembered road class colors. A Bayesian classifier takes a d-dimensional measurement vector, $\mathbf{x}$, and chooses the best class label, w_j, from a set of K classes, using a previously computed, *class conditional* probability, $P(\mathbf{x}|w_j)$, for each class [4, Section 2.8]. For our case $\mathbf{x}$ = [*Red Green Blue Texture*]^T. We assume that the *class conditional* probability can be modeled by a Gaussian distribution and therefore, is totally specified by $\{\mathbf{m}_j, C_j, N_j\}$, the mean color and texture, the covariance matrix describing the relationship between the colors and texture, and the number of samples in class w_j. This classifier can be shown to be equivalent to picking the class that maximizes the following likelihood:

$$\lambda_j = \ln(N_j/N) - d/2 \ln(2\pi) - 1/2 \ln(|C_j|) - 1/2\ (\mathbf{x}-\mathbf{m}_j)^T C_j^{-1} (\mathbf{x}-\mathbf{m}_j)$$

where each pixel provides a four dimensional measurement vector (d = 4). To get the *Probability Image* value the exponential function is applied to the maximum likelihood value. The sign is negated if the maximum class is an off-road class.

This module was implemented in W2 by once again dividing the input 30×32 *RGB Images* and *Texture Image* into column stripes. The input statistical color models were duplicated on each cell. Notice that the first three terms of the likelihood calculation can be computed only once for each class rather than once for each pixel and was passed as input into the Warp array. To get the desired probability measure, an exponential function is needed. An approximation to an exponential was implemented on the Warp cells.

4.4.2.3 Road Hough

This SCARF module searches through all possible road interpretations for the road having the greatest accumulated probability based on the *Probability Image* from the *Classifier*. We assume the road is locally nearly straight, and can be parameterized using (v, θ) where v is the column where the center of the road intercepts with the vanishing row in the image and where θ is the angle difference from perpendicular where the center line lies (see Figure 4.3.) These two parameters are the axes of an accumulator space used for collecting votes. Each pixel in the probability image votes for all the roads that contain that pixel by adding its probability to the proper positions in the accumulator. For each angle θ_i, a given pixel location (r, c) will vote for a set of vanishing points lying between v_s and v_e given from the equations below:

$$v_s = c + (r - horiz)\tan\theta - (w/l)(r - horiz)$$

$$v_e = c + (r - horiz)\tan\theta + (w/l)(r - horiz).$$

where *horiz* is the horizon row in the image, w is the road width at the bottom of the image, and l is the length from the horizon row to the bottom of the image. The maximum value of the accumulator is chosen to be the road.

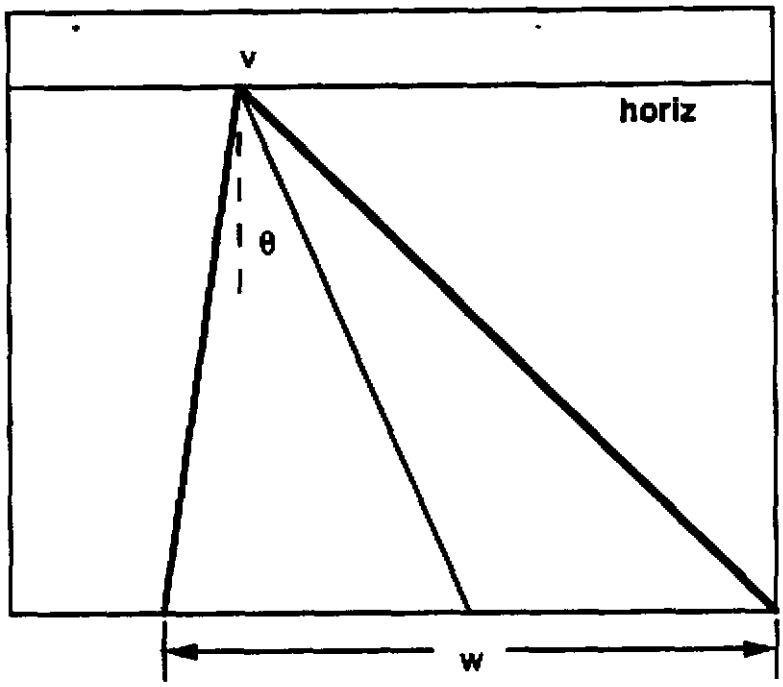


Figure 4.3: Road Hough

This module was implemented by distributing the input *Probability Image* column-wise among the cells. There was no overlap of the input between cells. However, a complete Hough space was calculated on each cell. To get the output Hough, the individual cell's Hough spaces were added as the output Hough Space was passed through the array.

4.4.2.4 Color Model Generator

This function calculates the road and off-road color models after each image is processed so that the system can adapt to changing illumination conditions. The texture model is not adapted and therefore is not computed after each image like the color model. The road and off-road color models are modified in three steps. First a set of pixels in the current image is chosen as road and off-road training sets. Next the training sets are subdivided into classes using an ISODATA clustering algorithm. Finally, the new statistical color models are calculated for each class.

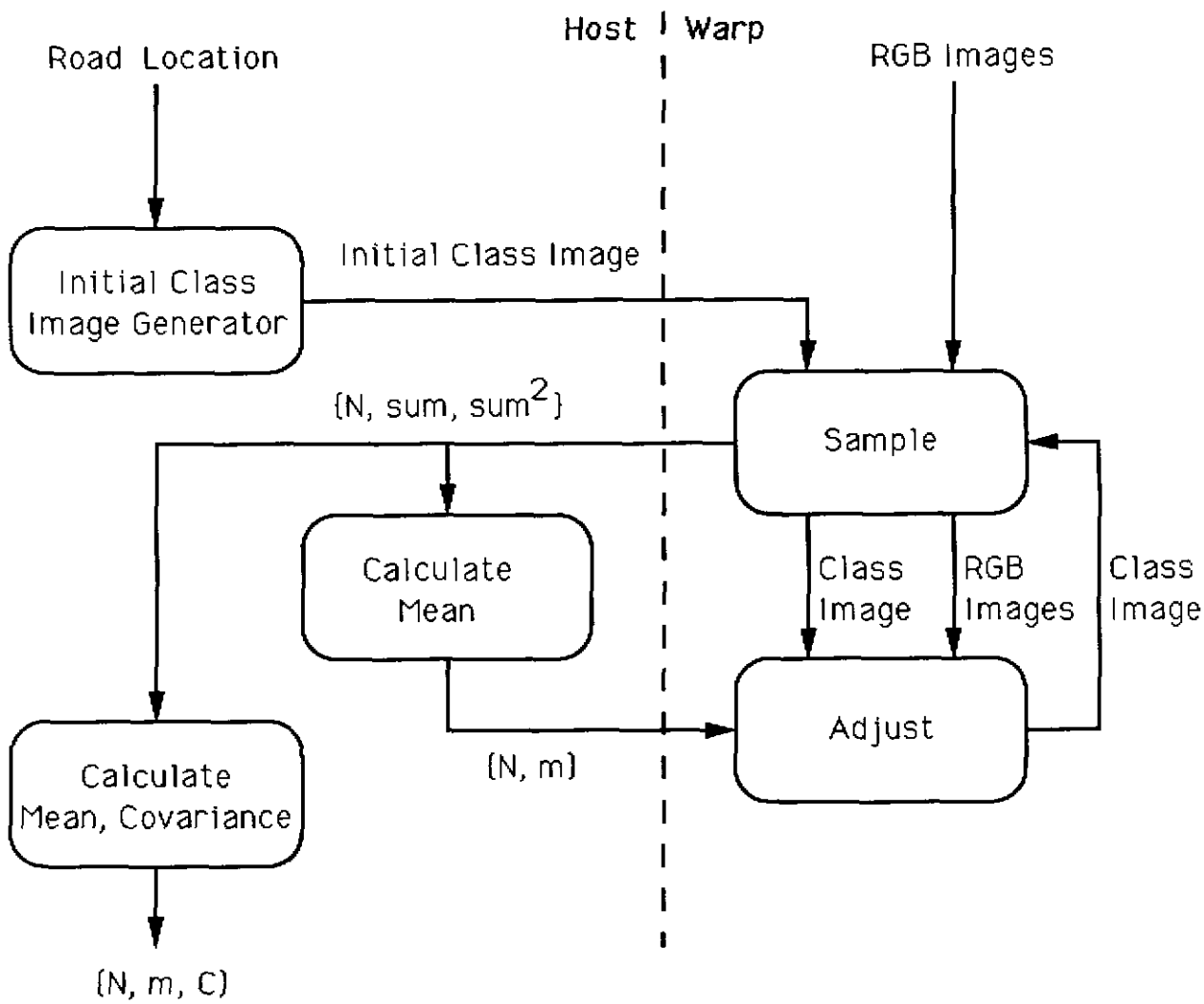


Figure 4.4: Implementation of ISODATA Clustering on the Warp Machine

The ISODATA clustering algorithm was implemented by a pair W2 modules; the *Sample* module and the *Adjust* module as shown in Figure 4.4. It started with a *Class Image* generated on the external host and the input *RGB Images*. This pair was called several times before the new color models were formed. After the last iteration, the *Sample* was executed one last time to generate the sums required for the final color models to be calculated.

The Sample Module computed sums of pixel values from a labeled input training set. The labels for each image

pixel were stored in the input *Class Image* which either labeled each pixel as one of the road or off-road classes, or as unknown. It also read the *RGB Images*. From this it accumulated the sums, $\mathbf{sum}_j$, and the squared sums, $\mathbf{sum}_j^2$ of the red, green, and blue pixel value for each class w_j. It also counted the total number of samples per labeled class, N_j. Color values were calculated for each class using samples labeled from the *Class Image*. From this information, the mean color of each road and off-road classes were calculated by an external host routine.

The Adjust Module adjusted the *Class Image* by using the current mean colors road and off-road classes. It read the mean class colors from the external host, the old *Class Image*, and the input *RGB Images*. Any pixel that was labeled as unknown in the old *Class Image* remained labeled as unknown. If the pixel was labeled as one of the road classes, then the pixel color (from the input color images) was compared with each of the road mean colors. The pixel was then re-labeled as the class whose mean color most closely matched the pixel color value. Similarly, if the pixel was labeled as off-road in the *Class Image*, then the new *Class Image* label was determined by the class whose color mean value was closest to the original pixel data. This module wrote a new *Class Image*.

The Sample module was implemented in W2 by dividing the *RGB Images* and the *Class Image* evenly among the cells. Each cell calculated its own partial sum of the color values for each class, partial sum of the color values squared, and number pixels with each label. The resulting sums were accumulated as the values were passed out of the Warp array and to the external host. The host then calculated the statistical color models using this values by the standard statistical equations for mean and covariance. The new mean values were then passed into the Adjust model.

The Adjust module was implemented in W2 by also dividing the input *Class Image* and *RGB Images* evenly column-wise among the cells. The mean values were copied to all of the cells. Therefore each cell produced a column stripe of the output *Class Image*.

4.4.3 Performance of SCARF Implementations

SCARF was picked for implementation on the prototype Warp machine in January of 1987. At that time, the system was already implemented on a Sun 3 and was processing images in about thirty seconds per image. During the time that W2 code was implemented, the C code was optimized, giving a final Sun 3 time of about twenty seconds.

The first implementation of SCARF used the prototype Warp machine and was completed in March of 1987. This implementation used only eight of the ten available cells so that the column data could be distributed evenly among the cells to simplify the implementation. Initially, the modules were implemented as described above. However, the time for downloading microcode to the Warp cells and the time required for passing data to and from the cells prevented any speed up of the implementation on the Warp machine over the Sun 3 implementation. This implementation required about fourteen downloads of Warp microcode and image data.

To improve the processing speed, the microcode for individual modules were linked together forming three microcode blocks. As a result, the microcode was downloaded only three times per image. To improve this rate, we implemented the two Roberts operators in W2 to reduce the size of the microcode required for these modules. At this time, the microcode could be linked into two separate sections and then required only two downloads per image.

We also noticed that we were passing in input image data repeatedly to the Warp array. By locking the Warp machine so that no other users could access the machine, and declaring the repeated input as global data in the same position in each W2 module, the input could be loaded once, and then used by different modules without being reloaded. With this modification we started to see an improvement over the Sun 3 times.

We implemented a version of *Image Pyramid Generator* on the cluster processors. This removed some of the load from the Warp array and had the additional advantage that the data from the frame buffer did not need to be copied to the external host before it was transferred to the cluster memory. Instead, the reduction implementation read the first of its inputs directly from the frame buffer and this data was never copied. In order to be as fast as possible, however, this implementation only approximated the averaging that was done in the Sun implementation.

Using microcode linking, common global storage, and by hand optimizing the W2 code, we were able to get the system running in about ten seconds per image. This speedup was small, but the results were promising for future implementations on the production Warp machine.

The next implementation of SCARF was more computationally expensive than the original version. It used two color camera inputs rather than the original one camera. It also classified 60×64 images rather than the original 30×32 images. The image pyramid generator now had twice as much data to process, and the classifier and the color model update had four times as much data since the vectors were now six dimensional rather than four as before. This implementation ran in about sixty seconds on a Sun 3.

This new SCARF was implemented on the new production machine by Chen and Crisman. On this machine, we had eight times more cell memory for global data and for programs. This machine used DMA for faster I/O from the external host to the Warp cells. The image data was now divided by rows on the cells rather than by columns. This allowed an even division of the 60 rows among ten cells rather than 64 columns divided among eight cells.

The new machine allowed successful use of the reverse data path feature where data could be accumulated on one data path onto the last cell, then the data could be passed back and copied into the other cells. This then allowed us to combine the *Sample* and *Adjust* modules from the Color Model Update into one W2 module. To do this the *Sample* module was modified so that after all of the sums were passed to the last cell, the last cell would calculate the new mean values and pass then back to all of the other cells. Then the *Adjust* module can be run without the intervention of the external host.

The larger memory of this machine also allowed us to store the *RGB Images* in a global memory location and then only input this data once per processing step. The increase in program memory allowed all of the modules to be linked into one microcode which was only downloaded to the Warp machine once before any image processing began. This implementation required about ten seconds per image, which was a speedup of six over the Sun 3 implementation.

The last implementation of SCARF on the Warp machine was completed in September of 1989 on the production machine. This version still used 60×64 images; however, it returned to the original one camera version of the code. The entire SCARF loop was implemented in one W2 function and one function was set up to initialize the whole system. The initialization process read in a 480×512 image and created a 60×64 image in Warp's cell memory. The W2 SCARF loop function started by doing the *Color Model Generator* on the resident image in memory. Then the new color image was read into the Warp cells and reduced. Next the *Classifier* and the *Road Hough* was applied to the input image. Only the resulting road location was passed out of the Warp array. Therefore, the main W2 loop function read only the full size color images, and wrote a couple of floating point numbers representing the road location in the image.

This version of SCARF ran in one second of Warp machine time, and a total of three seconds of time which include some limited displays and sending motion commands to the robot vehicle. This implementation is challenged only by a similar Sun 4/Androx implementation which processes lower resolution 30×32 images in 3.5 seconds.

4.5 ALVINN

ALVINN (Autonomous Land Vehicle in a Neural Network) applied connectionist techniques to the same problem addressed by SCARF, that is, road following using color images [13]. The key difference is that while SCARF was "trained" by hand, adapting standard vision algorithms to the recognition of a road, ALVINN used a neural network learning algorithm to automatically learn what image features were useful to discover the position of the road.

The development of ALVINN on the Warp machine went through two phases. In the first phase, from approximately February through May 1989, a road image generator was implemented and used to generate training images that were fed, together with the correct road position, to a standard back propagation learning algorithm. The back propagation algorithm ran on Warp, off-line; training runs were done overnight in the lab. After training, the learned network was used on NAVLAB to control the vehicle. (The network was run on a Sun 3, since application of the network, once it was learned, required relatively little computation).

This training technique fully exploited the power of the Warp machine; eight hour runs were used of the Warp machine, with approximately three-quarters of the time during these runs representing actual Warp machine time. Comparable training on a VAX 780 would have taken months.

Training using this method demonstrated the feasibility of using a neural network to control a robot vehicle. But the method suffered from a serious problem. Essentially, the process of adapting computer vision techniques to road recognition was replaced by the process of adapting computer graphics techniques to road image generation. This "forward" generation problem was easier than the "inverse" recognition problem, at least for the simple roads in the park, but it still required human intervention, so that the generated road images accurately represented the range of images that would be presented to NAVLAB. For successful navigation in more varied environments, the road image generation code would have to become more and more complicated and difficult to program and test.

To overcome this, training "on the fly" was attempted, starting in June 1989. Road images were taken directly from the camera, reduced in size, and presented to the neural network together with the current driver's steering angle (with the vehicle under human control). A clever technique was used to create many example images from one road image and steering angle. With the Warp machine on NAVLAB, backpropagation was used to modify the neural network weights as the vehicle was driven up the road.

Remarkably, it was found that with a short sequence of a few tens of images, the network could be trained successfully to follow the road. The eight-hour runs on the Warp machine in the lab were replaced by short runs driving NAVLAB for about ten minutes. Apparently, the intense training of the network in the long runs was unnecessary; in fact, the road following problem was much easier than it had appeared based on the long runs.

4.6 Evaluation of the Warp machine on NAVLAB

We now critically evaluate the Warp machine in light of the NAVLAB experience. We will treat hardware and software separately.

4.6.1 Warp Hardware

The Warp hardware consists of three components: the Sun host, the external host, and the Warp array itself.

4.6.1.1 The Sun Host

The choice of a Sun as the host of the Warp machine was one of the good early decisions made in the Warp project. At the time, the Sun workstation was one of the most powerful general-purpose workstations available; as it turned out, Sun continued to lead the field both in hardware and in software. The NAVLAB group decided to use Sun workstations as the basic general-purpose computing element on NAVLAB. Since the Warp machine had a Sun host, NAVLAB programs could be run on the Warp host whether or not they used the Warp array. This was important because of the limited space and power on NAVLAB; having to provide power and space for a workstation that could only be used for controlling Warp programs would have been an extra burden. This is demonstrated, in fact, by the eventual decision to remove the Warp machine from NAVLAB; this happened largely because the NAVLAB group moved to Sun 4 workstations. Upgrading the Warp host to a Sun 4 would have required extensive changes to the software. Thus, the Warp host would have been unavailable for running the rest of NAVLAB software.

4.6.1.2 The External Host

The Warp machine's external host consists of three MC68020 processors in a VME card cage, together with their memories totaling fourteen megabytes. Two of the processors input and output data to the Warp array, through a special board called the switch; these processors are called the "cluster" processors. The third processor performs auxiliary functions; it is called the "support" processor. The external host communicates with the Sun through a VME bus repeater. The external host card cage also held commercial digitizer boards, which were originally Datacube and later Matrox VIP boards.

A key early decision was to use a commercially available system that was programmable in C. This decision has been validated by the ease of code generation by the W2 compiler for the external host input and output routines (using a commercially supplied C compiler) and by the availability of commercial boards for digitization. We had to do little software development for the basic functionality, and hardware development was limited to the switch board.

Because of the use of industry-standard processors and busses, the external host was the weakest part of the Warp machine. In our early versions of FIDO on the Warp machine, this kept us from realizing full use of the Warp array, because of the constraints in rearranging data on the external host.

The decision to include the support processor was questionable. The support processor was intended to be used for auxiliary functions, such as controlling the digitizer board and possibly controlling the driving functions on NAVLAB; as it turned out, the digitizer boards were controlled by the Sun, and NAVLAB driving functions were controlled by a separate processor entirely. In fact, the support processor was never used, because of the difficulty of programming it and the absence of almost any debugging facility in the external host. System cost could have been reduced by almost a third by eliminating the support processor, with no loss of functionality.

Many of the external host capabilities were completely unused. For example, it was possible to use the external host to drive an RS232 connection; this connection, or another similar standard interface, could have been used to control NAVLAB. This was never done, because the NAVLAB controlling software and hardware was developed independently, and because of the difficulty of adapting such an interface to a new computer like the Warp machine.

Placing the digitizer boards in the external host was also questionable. The intention was to feed data directly from the digitizer boards to the Warp machine under control of the cluster or support processors. In fact, the normal method was to copy data from the digitizer board into the Sun's memory, and then to subsample the data there and pass it to the cluster processor for use in the Warp machine. Only in the most optimized versions of FIDO and SCARF did we actually use the support processor to take data directly from the digitizer board. In all other systems,

the data traveled twice over the VME repeater connecting the external host to the Sun.

The large memories in the external host were rarely used in NAVLAB. NAVLAB datasets were generally quite small, and there was no need for more than a few megabytes of memory. However, the large memories were useful in order to maintain compatibility with the Warp machines in the lab, where the large memories were used by other programs. When NAVLAB was docked and connected to the Ethernet, programs could be run interchangeably on the NAVLAB or the lab Warp machine.

4.6.1.3 The Warp Cell Array

The overall structure of the Warp cell array, a short linear array, has been validated by our experience on NAVLAB. The linear array was quite capable for the low- to mid-level vision algorithms we intended to implement on it at the start of the project; as the range of applications increased to include mid-level processing in SCARF and the neural network back propagation algorithm, the same linear array was usable. The key reason for this was the very high I/O rate within the array; this allowed us to overcome its limited connectivity.

The short linear array also lent itself to dealing with the relatively small datasets (32×32 or 64×64 images) in SCARF. Our early applications studies of the Warp machine were oriented towards dealing with standard 256×256 or 512×512 images. We thought that the increased power of the Warp machine would make it possible for the same processing then being done on small images to be done on large images, which would improve the accuracy and utility of color vision. As it turned out, this was not true. No increased vision performance could be obtained by using more spatially dense images; the road was a fairly large object in most of the scene, and where it was small (near the horizon), recognizing it accurately was useless, because of other uncertainties in the system. So we turned instead to processing small images with the Warp machine. The short linear array was just as suitable for this as it was for processing large images; in fact, with small images the relatively small Warp cell memory could be used to store previous images and other datasets, as in SCARF.

The two-way I/O pathway within the array was used to allow the computation of a result known to all cells entirely within the array; this was used in SCARF and in some implementations of the back propagation algorithm. In fact, for many purposes a circular connection (allowing the last cell to communicate directly with the first) would have been preferable.

The Warp cell included hardware floating point; this facility was one of the main reasons for building the Warp machine in the first place, and was one of the most expensive features in the Warp machine. The NAVLAB application made good use of Warp's floating point hardware. In SCARF, floating point was used extensively in the calculation of road statistics and their application to color pixel classification; in ALVINN, floating point was necessary for a good implementation of the back propagation algorithm. These applications of floating point came from outside the Warp project; independently, the NAVLAB group began using statistical methods for color classification, and the neural network group required floating point for their work. Without floating point the Warp machine would have been far less effective as a tool on NAVLAB.

4.6.2 Warp Software

As with hardware, we divide the software discussion into three parts: Warp host (Sun) software, external host software, and Warp cell software (the W2 and Apply compilers).

4.6.2.1 Warp host

The Warp array was used as an "attached processor" to the Sun. Datasets were downloaded into the external host, and then the Warp array was called to process them, usually while the Sun waited. (In fact, Sun processing could go on in parallel, and this was done in some of the SCARF systems. But generally this feature was not exploited because there was little for the Sun to do from the time the image was captured to the time the road was recognized).

This model was extremely useful in the development of software for the Warp machine. It was implemented using a mechanism that allowed replacement of a subroutine call in C by a single subroutine call in the Warp software package; the subroutine handled all transfer of data to and from the Warp external host, locking the Warp machine for exclusive use, and downloading and call of the Warp program. This could happen even if the Sun executing the call to the Warp machine was not a Warp host; data would be transferred over the Ethernet to a selected Warp host. It is quite likely that the Warp machine would not have been used much at all in real applications without such a simple method for accessing the machine.

However, the attached processor model implies many overheads. The Sun can become a bottleneck for processing. The startup time for the Warp machine can be quite significant. A serial processor, the Sun, must prepare datasets for a much more powerful parallel processor, the Warp array. Data structures must be moved from the Sun into the external host for processing. All of these overheads seriously affected the performance of the Warp machine on NAVLAB.

For example, images as captured by the Datacube boards were 480×512 in size. They had to be reduced in size for processing, since spatial resolution was not an important factor in road recognition. This could be done on the Sun, in the external host, or on the Warp machine. Existing libraries of software made image reduction on the Sun trivial – in fact, transparent to the programmer. Programming the external host (the logical place) to do the reduction was difficult, and the Warp cell array could do the reduction only if the images were first transferred from the Datacube boards to the external host memory either by the Sun or the external host. These tradeoffs made image reduction usually happen on the Sun, although in some SCARF systems it was implemented on the external host.

The Warp machine's startup time (time to start up a Warp program with the code already downloaded) was about 25 ms. This time was not a significant fraction of processing time for 256×256 or 512×512 images; in fact, the minimum processing time for 512×512 images was about 60 ms, and usually several times longer. But for small, 32×32 or 64×64 images, this time could be a significant fraction of total time, particularly if several Warp functions were applied to process the image and recognize the road, as in SCARF. As a result, the programmer had to spend a lot of time organizing the Warp functions so they could be executed as the result of a single call, to reduce the overhead. We would have been better served had the startup time been significantly reduced; this could have been done by providing special hardware in the Warp machine's interface unit to allow the Warp machine to initialize itself. As it was, the Sun had to issue special commands to do the various stages of initialization.

Moving data structures back and forth from the external host implied considerable programming difficulty, since many of these structures were embedded in various ways in C programs. This was especially true for FIDO, an old program that had been worked on by a number of programmers. All of the data structures had to be "cleaned up" before the Warp programming could begin; and the process of cleaning up the data structures introduced new overheads.

If we had not used the "attached processor" model, we might have taken an "array-centered" view of the Warp machine. In this model, the Warp array would have been viewed as the central processing resource, and other devices, such as the external host, would have been viewed as supplying data for the Warp machine. We could have

attached multiple I/O devices, supplying data from different cameras and perhaps the ERIM laser scanner, and coordinated the processing through the Warp machine. This model would have required much more sophisticated Warp software; we would have needed an operating system on the Warp array to manage all these resources. But such a model would overcome the other problems discussed in this section.

4.6.2.2 External host

The external host software was designed on the assumption that in NAVLAB speed was of overriding importance, code would be linked together before runtime (i.e., runtime downloading of code was not necessary), and a library of compiled code could be built up that was not changed frequently.

In fact, these assumptions were largely untrue. The W2 compiler made it possible to write new routines and test research ideas using the Warp machine, which made it much more important to allow rapid testing of new code. (The early development of FIDO, with its programming of a few routines in microcode by several programmers over a period of months, much more closely matches the model we had in mind when the external host software was designed.) The difficult programming environment of the external host, which might have been acceptable if the code was not modified much, instead meant that it was reduced to performing I/O to and from the Warp array, using programs generated by the W2 compiler. And testing of new Warp routines could be best done using an interactive system, which meant that the external host software had to be adapted to allow runtime downloading of code.

One of the reasons for the difficult programming environment on the external host was that control of the development of external host software was transferred to Carnegie Mellon's industrial partners at an early stage of the project, well before it was used. This made it difficult to change the software as our applications experience grew. The industrial partners did a competent job of maintaining and extending the external host software as it was originally designed; but the software would have had to be redesigned extensively to be widely used in NAVLAB.

The use of the external host in FIDO shows its capabilities when the programming difficulties are overcome. Irregular operations were mapped onto it as part of pre- and post-processing of data from the Warp machine. Also, it performed memory access-intensive but not compute-intensive computations as well as or better than the Warp array, which could also allow the Warp array to be used for something else in the meantime.

4.6.2.3 Warp array

In this section we discuss the Warp array software (primarily the W2 compiler) as seen by the user. This includes many design decisions that were essentially forced by the Warp cell hardware, and thus are really hardware issues. Distinguishing between W2 issues forced by the hardware and forced by other concerns is appropriate for another report.

W2 made the Warp machine much more programmable than we expected. This led to major changes in the importance of some parts of the system and made it possible to overcome deficiencies in one area by using another instead. For example, we could modify our programs to accommodate a regular data pattern from the host, which led to higher I/O rates. This was important even in the later versions of the host, which had faster processors and higher data rates, but which could use DMA, which required a regular address pattern. This flexibility was the main reason we were able to observe the predicted performance of FIDO in actual Warp runs.

W2 is a simple "Pascal-like" language for programmers to implement. All that is required is that the programmer understand a very simple model of the machine, i.e. that there are 10 cells in parallel connected by a data path. The programmer, however, must decide how to parallelize his programs. A W2 function to average a 4x4 window of pixels is as follows:

```
procedure reduce();
begin
    int r, c, row, pos;
    float acc;

    for r := 0 to eval(SWATHROWS-1) do begin
        for c := 0 to eval(NCOLS-1) do begin
            pos := 4*r*IMGCOLS+c*4;
            acc := imgbuf[pos]+imgbuf[pos+1]+imgbuf[pos+2]+
                imgbuf[pos+3];
            pos := pos + IMGCOLS;
            acc := acc+imgbuf[pos]+imgbuf[pos+1]+imgbuf[pos+2]+
                imgbuf[pos+3];
            pos := pos + IMGCOLS;
            acc := acc+imgbuf[pos]+imgbuf[pos+1]+imgbuf[pos+2]+
                imgbuf[pos+3];
            pos := pos + IMGCOLS;
            out[r*NCOLS+c] = pos * 0.0625;
        end;
    end;
end;
```

Before this function is called, the input image to be reduced is divided into row swaths across the cells. 'Pos' marks the position in the input image and 'acc' accumulates the sum of the pixel values.

W2 made it possible to experiment with different algorithms, in the context of a research system such as FIDO, while getting good use of the powerful Warp array. As we programmed more and more of FIDO on the production Warp machine, programmability was essential, especially as it allowed us to make use of more complex programming models that used the powerful Warp array more and required less intervention by the relatively weak host.

Apply was used far less in the NAVLAB work than we had hoped. This was partly because of the relatively late introduction of Apply into the project; the first true Apply compiler for the Warp machine was not running until the fall of 1987, one year after the NAVLAB group began working with the Warp machine. By this time much of the programming difficulties Apply addressed had been overcome by learning on the part of NAVLAB programmers. Just as important, Apply code tended to be larger than W2 code for the same problem. In order to process the borders of the image properly, Apply duplicated the inner loop of the image processing function once, leading to a doubling of the code size. This was a serious problem when the user was attempting to keep all code for the entire SCARF application, for example, on the machine at the same time. W2 programs were smaller, though no faster than the Apply programs.

Border processing was a problem for the W2 programs, too, however. The C functions processed borders by duplicating rows and columns near the edges of the image. This was hard for the W2 programs to do. It was simpler just to use a constant value (0) for the border of the image. However, this led to spuriously high values of edge detectors like Roberts near the image border. This sometimes affected the accuracy of the road image processing based on the texture image.

W2 did not support function calls until quite late in the project. Macro calls were used instead. This led to code size problems for transcendental functions, so that approximations were used instead. This often led to less accurate results than those used on the Suns. In particular, the classification in the SCARF system was often noisier for the W2 implementation than for the Sun implementations.

There were some peculiarities in converting data between the external host and the Warp array. Because the

Warp machine's primary processing power was in floating point, all images were best processed in this way. The interface unit had hardware conversion of 8-bit and 16-bit integers to and from floating point, but could not convert 32-bit integers. As a result, 32-bit integer images in the external host had to be treated differently from 8- or 16-bit integer images.

Primarily as a result of the fixed-size queues, it was impossible to use variable-length for loops in W2 programs until the design change that allowed blocking on writing to a full queue, or reading an empty one, was fully integrated into W2. This meant that image sizes were fixed at compile time. This increased code size (for example, in SCARF there were three different versions of the Roberts operator).

It was impossible within W2 to execute a W2 function repeatedly until some condition was met, for example repeatedly classifying image regions until convergence was achieved. This was a consequence of the distributed nature of control in the Warp machine. The result of this was that the Warp host was involved in repeatedly calling W2 programs and testing for convergence, which significantly increased overheads.

Partly as a result of the small datasets used on NAVLAB, and partly as a result of limited hardware support, we still needed to use speed tricks to generate Warp code that was significantly faster than the Sun code. For example, we had to avoid using division on the Warp machine, especially when doing integer index calculation for arrays.

Given the research that was going on in the Warp project while the Warp machine was being used in the NAVLAB project, it is remarkable that things worked as well as they did. NAVLAB programmers commonly had to deal with new features in the W2 compiler, for example, and it is only due to the good support from the compiler group that they were able to overcome bugs that were due to idiosyncrasies of the machine, and were extremely difficult to identify without experience.

4.7 Conclusions

This report tells the story of a unique experiment; the installation and use of a parallel supercomputer on a robot vehicle. Let us try to summarize and draw some conclusions from this experiment.

- The Warp machine was useful in the NAVLAB project. The programmable floating point capability it brought to NAVLAB was unavailable by other means. Key elements of the architecture, such as its high I/O rate and the short linear array, have been validated by the NAVLAB experience. The high processing rate of SCARF could not have been obtained without the Warp machine, and it was the presence of the Warp machine on NAVLAB that led to ALVINN.
- Early applications support is essential. The development of FIDO and other vision applications guided the early design of the Warp machine and provided test programs and early demonstrations. Without this early work, the Warp machine may never have been used on NAVLAB.
- Continuing software support is essential in a project such as this. It is impossible for hardware and software designers to anticipate all of the issues that will turn up in use of the machine, even if applications designers participate early in the project. This is partly because applications can change, and partly because success in one area can affect others. For example, the Warp machine became much more programmable than we anticipated because of the W2 compiler, which made the design of the external host partly obsolete.
- The "attached processor" model used in the Warp machine is natural and easy to use by the programmer; but it leads only with great difficulty to large speedups in programs. Data structures must be redesigned, careful attention has to be paid to small details of implementation, and so on. If we want to see speedups more than a factor of about ten, we must abandon this model.

Acknowledgments

"We" in this chapter refers to a very large group of people, indeed; too many to list here. The Warp project at Carnegie Mellon and General Electric and Warp applications development (including the Parallel Vision and NAVLAB projects) included over seventy people. We also benefited from the support of the Field Robotics Center and from many discussions with the vision group at Carnegie Mellon.

Research on NAVLAB (including Parallel Vision research) was supported by the Defense Advanced Research Projects Agency, DOD, (DARPA) under contracts DACA76-89-C-0014, DACA76-86-C-0019, DACA76-85-C-0003, and DACA76-85-C-0002, all monitored by the Engineer Topographic Laboratories. The Warp project was supported in part by the DARPA under Contract N00039-85-C-0134, monitored by the Space and Naval Warfare Systems Command, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533. The ALVINN work was supported by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533, by National Science Foundation Grant EET-8716324, and by the DARPA monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies or the US government.

4.8 References

[1] Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A.
The Warp Computer: Architecture, Implementation and Performance.
IEEE Transactions on Computers C-36(12):1523-1538, December, 1987.

[2] Clune, E., Crisman, J. D., Klinker, G. J., and Webb, J. A.
Implementation and Performance of a Complex Vision System on a Systolic Array Machine.
Technical Report CMU-RI-TR-87-16, Robotics Institute, Carnegie Mellon University, 1987.

[3] Dew, P. and Chang, C.H.
Passive Navigation by a Robot on the CMU Warp Machine.
Aug, 1984.
Internal report, Department of Computer Science, Carnegie-Mellon University, Aug. 1984.

[4] Duda, R. O. and Hart, P. E.
Pattern Classification and Scene Analysis.
Wiley, 1973.

[5] Hamey, L. G. C., Webb, J. A., and Wu, I-C.
An Architecture Independent Programming Language for Low-Level Vision.
Computer Vision, Graphics, and Image Processing 48:246-264, 1989.

[6] Kung, H.T., Ruane, L.M., and Yen, D.W.L.
Two-Level Pipelined Systolic Array for Multidimensional Convolution.
Image and Vision Computing 1(1):30-36, February, 1983.
An improved version appears as a CMU Computer Science Department technical report, November 1982.

[7] Kung, H.T.
Systolic Algorithms for the CMU Warp Processor.
In *Proceedings of the Seventh International Conference on Pattern Recognition*, pages 570-577. International Association for Pattern Recognition, 1984.
A revised revion appears as Chapter 3 in *Systolic Signal Processing Systems*, edited by E. E. Swartzlander, Jr., pp. 73-95, New York, Marcel Dekker, 1987.

[8] Kung, H.T. and Menzilcioglu, O.
Warp: A Programmable Systolic Array Processor.
In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII*, pages 130-136. Society of Photo-Optical Instrumentation Engineers, August, 1984.

[9] Kung, H.T. and Picard, R.L.
One-Dimensional Systolic Arrays for Multidimensional Convolution and Resampling.
In Fu, King-sun (editor), *VLSI for Pattern Recognition and Image Processing*, pages 9-24. Springer-Verlag, 1984.
A preliminary version, "Hardware Pipelines for Multi-Dimensional Convolution and Resampling," appears in *Proceedings of the 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Hot Springs, Virginia, November 1981, pp. 237-278.

[10] Kung, H. T. and Webb, J. A.
Mapping Image Processing Operations onto a Linear Systolic Machine.
Distributed Computing 1(4):246-257, 1986.

[11] L.H. Matthies, C.E. Thorpe.
Experience with visual robot navigation.
In *Proc. IEEE OCEANS'84 Conf.*, pages 594-7. IEEE, September, 1984.

[12] Moravec, H.
Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover.
Technical Report CMU-RI-TR-3, Carnegie-Mellon University Robotics Institute, September, 1980.

[13] Pomerleau, D. A.
ALVINN: An Autonomous Land Vehicle In a Neural Network.
In Touretzky, D. S. (editor), *Advances in Neural Information Processing Systems*. Kaufmann, 1989.

[14] Thorpe, C.E.
FIDO: Vision and Navigation for a Robot Rover.
PhD thesis, Carnegie-Mellon University, December, 1984.

Chapter 5: Autonomous Navigation of Structured City Roads

5.1 Introduction

In 1985, 81.31% of the intercity passenger traffic in the United States — 1,418 billion passenger-miles — was done by private car. This translates into a tremendous amount of time spent by drivers engaged in the task of visually tracking and driving along a road. As autonomous road following programs become more competent they will be able to take over more and more of the burden of driving -- at first, in daylight under light traffic conditions, then later under more challenging illumination, weather, and traffic conditions. Driving long stretches on open freeway, while probably the easiest road following task to automate first, is only part of the larger domain of autonomous road following. The length of the average automobile trip in the United States in 1983 was just 7.9 miles [21]. In order to liberate people from the tedium of driving, road following systems will need to be able to follow city streets and maneuver through intersections, keeping track of what lanes are available for use, and not straying into lanes for oncoming traffic.

The gap between this vision of robot chauffeurs whisking people to and from work while they read the morning paper and the state of the art in robot road following is wide. While lane following on a freeway has been demonstrated at speeds of up to 96 km/hr [13], lane following is only one of the capabilities that an autonomous road following system must have. Current systems, while they have achieved fair levels of robustness in staying on the road, don't model the lane structure of the road. In order to progress toward the ultimate goal of robot chauffeurs, road following systems need improved capability to keep track of the lane structure of the road (both for purposes of lane following and for purposes of planning such as deciding if it is possible to change lanes or pass) and improved capability to detect and navigate through intersections.

Achieving these capabilities requires first the ability to robustly detect various road features (painted stripes, road/shoulder boundaries, etc.) under a variety of conditions (changes in lighting, pavement color, etc.). Dealing with different feature appearances can best be accomplished through the knowledge-based application of specialized segmentation techniques that work well in specific cases. As an example, yellow stripes can be located in both sunlit and shadowed image regions by thresholding on the pixel hue value. Given a known road geometry, the ability to reliably locate road features allows a system to determine the position of the vehicle relative to the road and drive the vehicle along the road in its lane. It is also necessary to instrument the segmentation techniques so that the system can detect when they have failed, and to implement strategies for determining the cause of failure and recovering from the problem. Lane boundaries may shift, requiring a change in the road model; markings may change in appearance, requiring a change in segmentation technique; or the vehicle may be approaching an intersection, requiring changes in sensing strategy.

Improvements in intersection navigation capability and the capability to detect and correct for changes in lane structure are particularly critical for making progress towards systems that can autonomously drive on city streets. On the highway, lane structure is relatively fixed, and the vehicle does not have to go around sharp corners or track the position of the road across a large intersection. In the city, lane edges shift as right and left turn lanes appear near intersections, and a vehicle needs to be able to maneuver its way through intersections. Intersections cover a large area on the ground, creating a need to combine information from several images in order to fix the location of the vehicle in the intersection and plan a path through it.

The problem of autonomous road following in an urban environment can be decomposed into the following subproblems: segmentation of the image data to extract road features, modeling of the local road geometry for vehicle localization and path planning, and intersection navigation. In the next section we examine representative existing systems, focusing on their approaches to these subproblems. After outlining our approach to these tasks, we

present results in robust detection of painted lane markings, fitting features positions to a model of road geometry, and locating road features without using a strong a-priori model. We close by discussing our planned extensions to enable the system to navigate through intersections.

5.2 Previous Work

5.2.1 VITS (Martin Marietta) [20]

- **Segmentation:** The road is extracted by thresholding a (red minus blue) image. The basic algorithm was extended to include two road classes, sunny and shaded, whose thresholds were found by sampling near the bottom of the image (which is assumed to be all road).
- **Road model:** The system assumes that the ground is locally flat, and projects the boundary of the road regions onto the ground. More sophisticated models of road geometry such as the hill-and-dale or zero-bank algorithm were rejected because of sensitivity to errors in segmentation and matching of corresponding points on the road edges, and because they could not handle intersections.

The VITS system was able to achieve fairly impressive performance, driving at up to 20 km/hr. on straight, obstacle-free stretches of road. While the paper referenced mentions intersection navigation as a criterion for selecting a technique for recovering the road shape, intersection navigation was not implemented. Sacrificing general capability for speed, the restriction to two road color classes limits the robustness of the segmentation.

5.2.2 FMC system [11]

- **Segmentation:** Regions from an Ohlander-Price style segmentation of an initial image are classified as road/non-road, and an optimal transform is derived to maximally separate road and non-road pixels. Normalized histograms of the transformed road and non-road pixels are used to generate likelihood ratios for each level of the transformed feature, which are then used to classify pixels in succeeding images. The likelihood ratios are updated for each image in order to handle changes in road color and lighting.
- **Road model:** Line segments fit to the road region boundaries are tested for continuity with the road boundaries from the previous image, consistency with constraints on the angle between successive segments on each side, and parallelism between segments separated by the expected road width.
- **Intersections:** The system examined the road region boundary segments for lines which might support an intersecting road.

The FMC system was also able to make runs at speed of up to 19 km/hr., with an image cycle time of 1.5 seconds. As in the VITS system, the road is reconstructed from the segmentation rather than fitting a road model to the results of the segmentation. Speed was a major criteria driving the design of the system, again resulting in a tradeoff between robustness and speed.

5.2.3 MARF (University of Maryland) [23]

- **Segmentation:** Small windows are placed at the predicted locations of the road edges at the bottom of the image. Sobel edge detecting and the Hough transform are used to determine the road edge location in the windows. The system then repeats this process, tracking the features up through the image.
- **Road Model:** Researchers at the University of Maryland investigated a number of algorithms for extracting 3-D road structure from image data. The most sophisticated algorithm [5] models the road as a horizontal segment swept perpendicular to a spine curve. Global optimization of the result is used to correct for errors in local point matching between the road edges.

The MARF (for *MAryland Road Follower*) system was ported to the Martin Marietta ALV and drove the vehicle. The algorithms for recovery of road shape from image data are probably the most significant contributions of this work. Recently they have been working on declarative visual search strategies for road following [6].

5.2.4 VaMoRs (UniBw Munich) [13]

- **Segmentation:** Six 48-by-48 pixel windows selected from a grey-scale road image are convolved with one of 16 oriented bar masks for edge detection.
- **Road model:** The system uses the flat earth assumption, and models the lane followed as having parallel edges with constant separation and locally constant horizontal curvature.

The VaMoRs system combines custom hardware for image processing with an elegant control formulation to achieve runs at speeds of up to 96 km/hr. The system fits the lane edge points to a model of the road geometry rather than reconstructing the road boundaries from the segmentation results. The system does not model road structure other than the lane being followed, and does not handle intersections. The experience of the NAVLAB group at CMU with the use of oriented edge trackers [22] suggests that reliance on them as the only method of segmentation will not be robust under difficult shadow conditions, although the Munich researchers claim that they have not encountered problems with this.

5.2.5 LANELOK (GMR) [8], [9]

- **Segmentation:** Several segmentation methods were tested. In the first method, edges segments are extracted from a thresholded Sobel edge image, and the edge segments vote in a Hough transform for right and left lane edges. In the second method, growing and shrinking are applied to the binary edge image to thicken the lane boundaries, and region tracing is applied to extract them. In the third method, search areas are defined around the expected locations of the left and right lane edges, and template correlation is done to find the lane markers. A least-squares fit is done to the optimum correlation values to determine the lane edges. These methods have been tested independently, and are not used cooperatively.
- **Road Model:** The lane is modeled as having parallel edges separated by a constant width. Shifts in the lane markers are detected and corrected for.

LANELOK's algorithms have been tested on more than 3000 frames of videotaped data. The system is designed to track lane boundaries in a freeway environment, and appears to work well, if slowly (three seconds/image on a VAX 8600 for the template correlation algorithm, similar times for the Hough algorithm). The system also incorporates obstacle detection, using template correlation to locate other vehicles in the lane.

5.2.6 *University of Bristol* [17]

- **Segmentation:** White lane markings are detected by creating a binary image in which the selected pixels correspond to pixels in the intensity image which are brighter than a threshold value and between two strong intensity gradients of opposite sign separated by the expected lane marking width. Regions in the binary image are extracted, and shape cues are used to eliminate noise regions.
- **Road model:** The surviving regions are backprojected onto the ground plane, and a parabolic model is fit to each candidate region. Dashed lane markings are accommodated by fitting arcs to all pairs of short regions. Minimum separation and constant separation constraints are used to eliminate erroneous candidate arcs, and to produce a final set of consistent lane markings.

From [17] it is unclear how much testing the algorithm has received, but the approach seems sound, and could adapt easily to use improved segmentation techniques.

5.2.7 ARF (CMU) [12]

- **Segmentation:** The system has two tracking algorithms, a profile correlation technique and a Sobel edge tracker. These algorithms are used in tandem to track roads in aerial images. If the results of the two tracking methods diverge, then failure analysis rules are invoked to determine the cause of the problem (intersections, changes in road width, changes in surface material, overpasses, occlusion, vehicles on the road), and appropriate corrective actions are taken.

- **Road Model:** The road is modeled as locally having a parabolic shape. No interior road structure is modeled.

The ARF system works in the domain of tracking road networks in aerial images rather than in a vehicle navigation domain, but is included because of the influence of its architecture on the design of YARF (specifically, the use of multiple segmentation techniques and explicit failure analysis).

5.2.8 Sidewalk II (CMU) [7]

- **Segmentation:** The system uses an earlier version of the color classification algorithm that is used in the SCARF system described below. It can also fuse the color segmentation with a range image segmentation to distinguish between stairs, a ramp, and the surrounding grass slope.
- **Road Model:** The system has a map of the geometry of the system of sidewalks it navigates on.
- ***Intersections:*** *Line segments* fit to the edges of the extracted road region are matched with expected edges from the map to determine position within an intersection.

The Sidewalk II system was designed to operate in an environment where the segmentation problem would be relatively easy, allowing exploration of the higher level issues of route planning and intersection navigation. It performed well, albeit at slow speeds, but is limited by its need for a geometric map of the intersections it will encounter.

5.2.9 SCARF (CMU) [4]

- **Segmentation:** An adaptive color classification scheme is used, with four to eight color classes each used to model road and non-road areas.
- **Road model:** The system assumes that the road *is locally straight, with a known constant width.* The classified pixels vote in a Hough scheme to locate the vanishing point and orientation of the road in the image.
- **Intersections:** Once the main road has been found, the pixels on that road are subtracted from the Hough space and further peaks corresponding to intersecting paths are searched for.

The SCARF system has been one of the most robust and successful of the road following systems developed at CMU, following the path up Flagstaff Hill under a wide variety of weather and road conditions. With its Hough voting scheme, it is the only one of the systems discussed that has an explicit representation of how certain it is that it has found the road.

5.2.10 ALVINN (CMU) [15]

- **Segmentation:** ALVINN does not have a fixed segmentation technique. It consists of a three-layer *backpropagation neural network* which is trained on the road to be followed or on simulated data.
- **Road model:** There is no model of the road, other than whatever model is implicitly encoded in the weights learned by the network.

ALVINN is *sui generis.* As mentioned above, it does not have a fixed segmentation and evidence combination strategy, but learns one from *training examples.* It performs very well, and has driven the NAVLAB at its top speed of 20 MPH. On the other hand, its lack of any explicit representation makes it hard to evaluate how general a road following capability it possesses (for instance, can it be trained to lane follow in any lane on roads of differing widths?)

5.2.11 Analysis

All of these systems use a single segmentation technique to locate the road, making them vulnerable to situations in which that technique fails. Binford made the same point with respect to object recognition programs [3]. Global color classification schemes such as those used in [20], [18], and [11] work well for segmenting the road surface from the background, but work less well at detecting painted lane marking because they look only at pixel color and fail to consider geometric constraints. Edge detectors have problems with textured areas and shadows, particularly mottled shadows from trees.

There are two classes of techniques used to compensate for errors made by the segmentation algorithms. The first is focusing, in which predictions of road feature location are used to limit the areas of the image which are examined [23], [13], [9]. The second is use of global constraints, in which a model of the road structure is used to eliminate errors in segmentation. A good example of this second approach is the Hough voting scheme used in SCARF, which uses the assumptions of constant known road width and a straight road to correctly locate the road even in cases where there are many misclassified pixels.

Few of these systems have any explicit representation of how confident they are that they located the road, making it possible for them to "hallucinate" and drive off the road. Only LANELOK (and ARF in the aerial road tracking domain) has any mechanism for detecting changes in road structure based on segmentation failures. Intersection navigation capabilities of these systems are very limited. This is largely because they process one image at a time, and real city intersections are large compared to the field of view of typical cameras. The exception to this is the Sidewalk II system, which used a second camera to see around corners at intersections.

YARF addresses these problems through the following mechanisms:

- multiple segmentation techniques which are specialized to detect particular kinds of features or to work in particular situations;
- examination of the results of the segmentation techniques and their geometric consistency with a model of the road structure to detect when the systems fails, when the road appearance or structure changes, or when the vehicle is approaching an intersection; and
- use of a local map to integrate feature location data and locations where segmentation failures have occurred over multiple frames.

The remaining sections of this chapter describe our current research in implementing these mechanisms, describing the progress we have made since the initial results reported in [10].

5.3 Robust painted stripe detection

A major component of the program of research we described in [10] was the investigation of specialized segmentation techniques to robustly extract different types of road features. Our recent experiments in this area have concentrated on testing two algorithms, one for detecting yellow stripes using pixel hue, and the other for detecting white stripes using an oriented bar detector. These algorithms have been tested both in open loop mode, where they track a stripe as a human drives the vehicle, and in closed loop mode, where their results are used to drive the vehicle. The implementation of these algorithms is described in detail in [1].

Hue appears to be a very stable cue for detecting yellow stripes under a wide variety of road and lighting conditions. Putting red at zero degrees on the color wheel, pure yellow has a hue of 60 degrees. Histograms of yellow stripe pixels in both bright, sunlit images and darker, shadowed images show a peak located at 60 degrees, with a width of 30 degrees on either side. Pixels with hues between 30 and 90 degrees are classified as yellow, pixels with hues outside this range are classified as background (see figure 5.1). In order to avoid grey pixels being classified as yellow due to the instability of hue near the intensity axis, we also require yellow pixels to have a saturation of at least 0.1. The algorithm does not explicitly compute the hue of the pixels. Instead, it tests the RGB

value against two planes containing the intensity axis which bound the desired section of the color cube. Those pixels whose RGB values fall on the correct side of both planes are labeled as yellow pixels, other pixels are labeled as background. The mean row and column of the yellow pixels is returned as the position of the center of the yellow stripe.

Robust detection of white stripes is done by looking for a bright bar of a specified width at a specified orientation. Using an oriented operator reduces the effects of noise such as shadows or oil stains on the pavement. Searching for a bar rather than an edge and blurring along the direction of the bar also improves the robustness of the operator. The correlation is done with the blue band of the color road image.

Two techniques are used together to achieve a fast correlation. The first is the use of only +1 and -1 as weights. This speeds up the correlation by reducing the number of additions and subtractions needed. When the mask is shifted one pixel to the right, the leftmost pixel previously included in the correlation sum is removed, the new rightmost pixel value is added in, and corrections are made for pixels whose weight changes sign when the mask is shifted. As an example, if the mask is (-1 -1 -1 1 1 1 -1 -1 -1), only four additions/subtractions are needed: one for the pixel which shifts off the left edge of the correlation window, one for the new pixel on the right edge, and one each for the two pixels whose weights change sign. The second technique used to increase the speed of the correlation is using a window which is a parallelogram parallel to either the rows or columns of the image rather than an oriented rectangle, which speeds up the correlation through a more regular pattern of pixel access.

Figures 5.2 shows these operators tracking the center double yellow line and the right and left white lines on a sunny, well-lit road. Figure 5.3 shows them tracking the center double yellow line and right white line on a road covered with mottled shadows from trees. While these algorithms do not perform perfectly, they appear to be more robust than any of the other techniques we had investigated. Detecting when these operators have failed to find the desired feature is simple. In the case of the oriented bar operator, the correlation peak will not differ sufficiently from the background level. In the case of the yellow hue operator, the area of the yellow pixel regions in the window is either very small (if there is no yellow stripe) or much larger than the road model would predict (if the window falls onto a grassy region — surprisingly, some grass has a hue very close to the hue of yellow stripes). In the next section we discuss the combination of the individual measurements of feature positions into an estimate of the local road curvature and the position of the vehicle on the road.

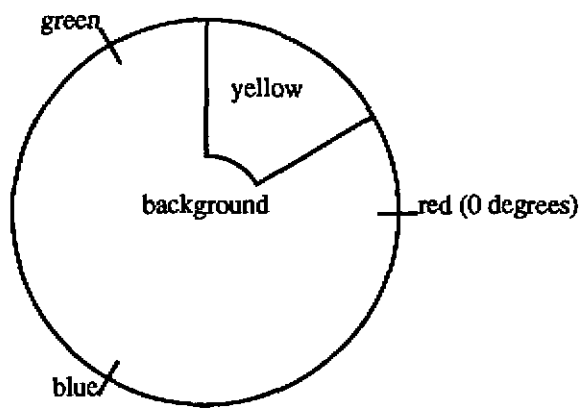


Figure 5.1: Color classification by hue to detect yellow stripes

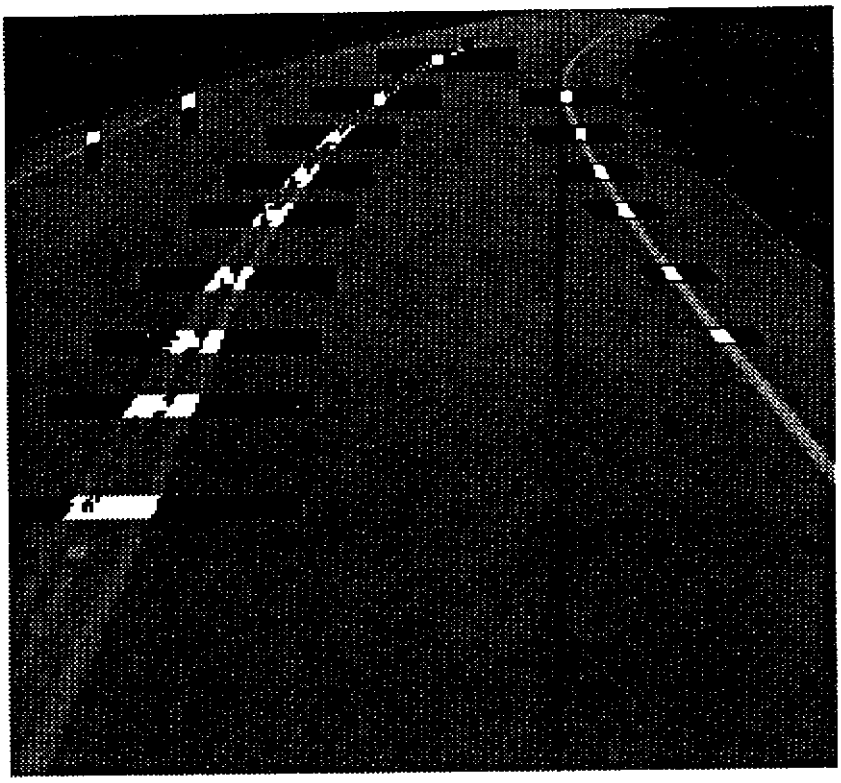


Figure 5.2: Yellow hue and white bar operators, sunny image

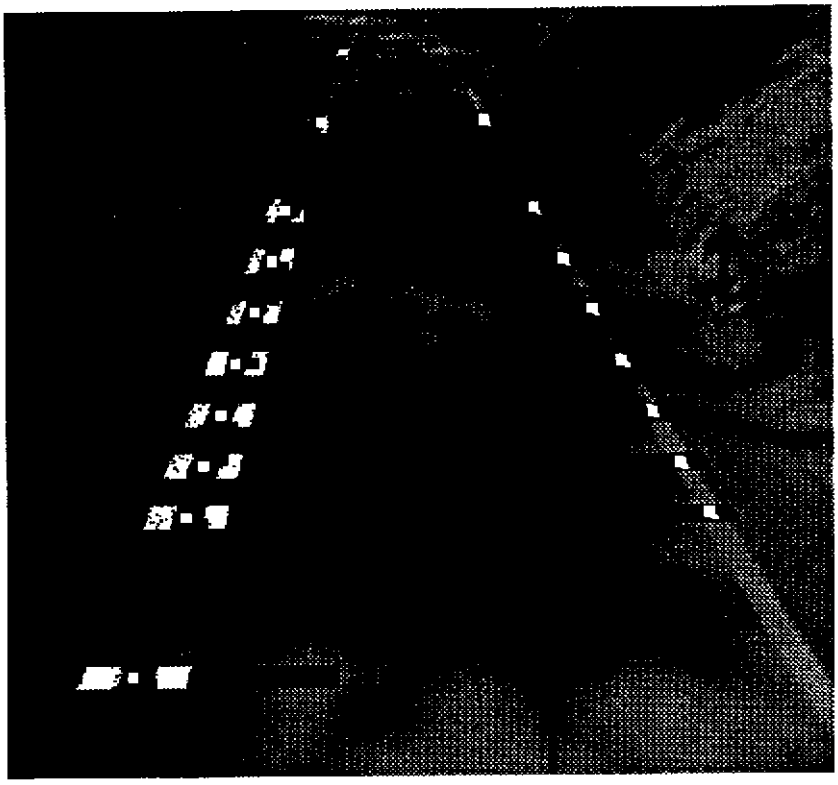


Figure 5.3: Yellow hue and white bar operators, shadowed image

5.4 The road model and fitting detected feature locations

YARF models the road as a *generalized stripe* -- a one-dimensional feature cross-section which is swept perpendicular to a spine curve. The road in figure 5.2, for example, can be modeled by the following feature cross-section:

- a solid white stripe which starts -358 cm. from the spine;
- a lane of pavement which starts -342 cm. from the spine;
- a solid double yellow line which starts 0 cm from the spine;
- a lane of pavement which starts 50 cm. from the spine;
- a solid white stripe which starts 403 cm. from the spine; and
- a shoulder which starts 419 cm. from the spine.

An important design decision is the question of how complex a spine curve should be allowed. Should the road model allow for banking? Should it assume a locally flat ground plane, or allow changes in surface slope? Answering these questions requires considering not only how roads behave in the real world, but what kinds of models produce algorithms that are computationally tractable and results which are stable in the presense of noise and usable for navigation even if they do not reproduce the world with complete fidelity.

We have chosen to adopt a model similar to that used in the Munich VaMoRs system [13] and work at the University of Bristol [17]. The road spine is locally approximated by a circular arc, with the road lying in a flat ground plane. In order to do a linear least-squares fit, a parabolic approximation to a circular arc is made, $x = 0.5 * curvature * y^2 + slope * y + lateral\_offset$. Such a model allows computationally efficient fitting, and produces results on real roads that allow robust navigation even though the actual road may not be flat or locally a circular arc. The model of the feature cross-section is used to correct the position of the detected points so that they lie roughly along the center spine of the road. We add the feature offset to the x coordinate given by the parabola equation above, which is a small-angle approximation to the proper correction (it assumes that the cosine of the angle between the tangent to the road and the y axis is approximately one).

Figure 5.4 shows the fit of the road model describes above to the feature positions detected in figure 5.2. The diamonds are the individual feature locations, labeled with the corresponding feature number. The black dots lie along the parabolic fit, and the equivalent circular arc spine road model is drawn to show the road features. To give an idea of scale, the tick marks on the line on the left of the drawing are spaced two meters apart.

In order to increase the stability of the parameter estimates, we fit the model to the points detected over the last several frames (typically three to six). Figure 5.5 shows data accumulated over a sequence of eight images placed into a global coordinate frame. The squares show the individual feature position estimates. The left digit of the number by each square is the frame number, while the right digit is the feature number in the cross-section model. The asterisks connected by dashed lines show the desired paths fed to the path planner, and the diamonds with lines pointing out from them show the estimates of road centerline position and direction. As can be seen from this figure, our camera calibration and the inertial navigation data produced by the vehicle are very accurate — the points from different images along the two lane markers line up very well as the vehicle goes through the gentle curve.

We have done some experiments with using robust M-estimation [2] to perform the fit. Standard least squares minimizes the square of the residuals of the data values. Robust M-estimation minimizes the sum of a function of the residuals which falls off more rapidly for large residuals, making the fit less sensitive to outlying data observations and allowing their detection. So far we do not have results which allow us to decide if this will successfully help in the detection of incorrect feature location data.

In the mode where YARF is following a road between intersections, a predict-segment-fit-move loop is used.

Inertial navigation is combined with the estimate of vehicle position from the previous image to predict feature locations in the current image. When started, the system does not have a prediction of where to place trackers to locate the road features. Initially we had a human operator use a cursor to select points along one feature to provide the system with the initial vehicle position relative to the road. Now we are experimenting with a technique to automatically extract candidate road features using Sobel edge detection, Hough transforms, and shared vanishing point and global continuity constraints. We describe this algorithm in the next section.

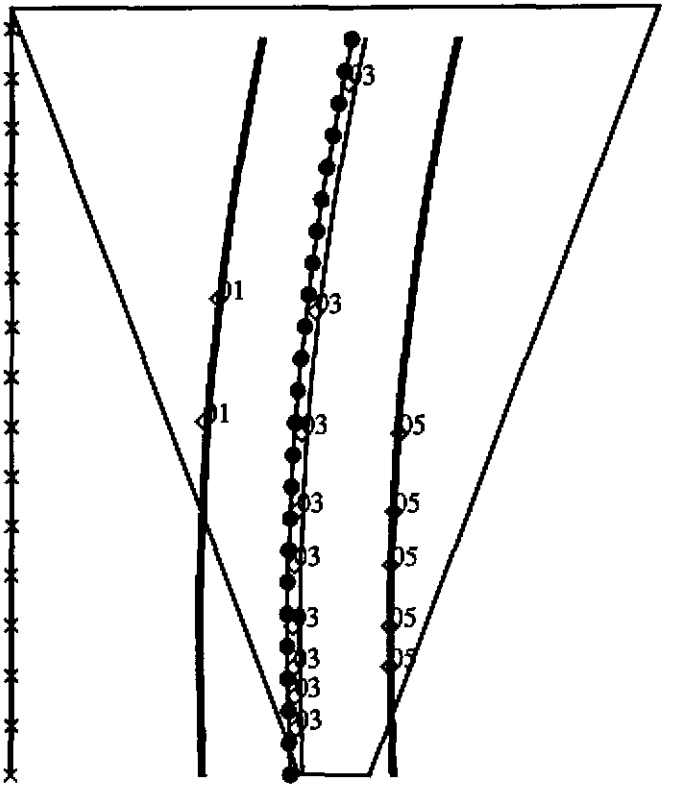


Figure 5.4: Fit of road model to detected feature positions

5.5 Bootstrap location of road features

There are two main techniques to compensate for incorrect segmentation results, the use of prediction to focus processing near features and the use of global constraint. In YARF's road-following mode, focusing is used, both to reduce computation cost and to reduce errors in feature location. In the absence of predictions of feature locations, global constraints must be used.

The features which form a straight road are parallel lines on the ground, which project into the image as lines which meet at a common vanishing point. In order to handle curved roads, the road is modeled as a sequence of straight segments by dividing the image vertically into a small number of horizontal bands and approximating the road as straight within each band, as Polk and Jain do [14]. In order to reduce the chances of noise in the image leading to the selection of a spurious vanishing point in some of the bands, a global optimization is performed which takes into account both the support for a given vanishing point within a band and the continuity of features between bands.

The Sobel edge detector is run on an image produced by preprocessing an RGB road image. We have experimented with various kinds of preprocessed images, including the red, green, and blue bands of the color image, the intensity image corresponding to the color image, and the (blue minus red) image. The gradient

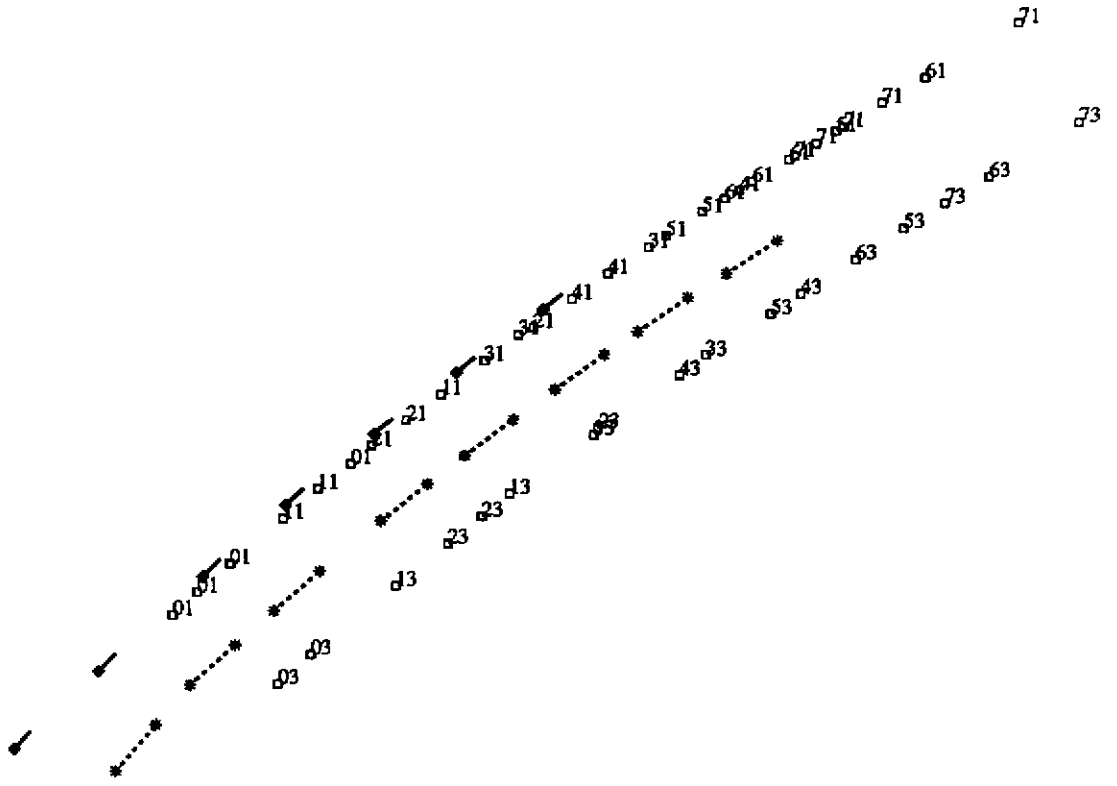


Figure 5.5: Feature locations in a sequence of eight frames

magnitudes are thresholded to create a binary image of candidate edge points. Any segmentation technique could be used which would give points where there are discontinuities in the image along with an estimate of the orientation of the discontinuities.

Lines in the image are represented by the column where they cross a specified row (in this case, the row which contains the horizon), and the angle they make with respect to the rows of the image. Given an edge point (*row*, *col*) with gradient direction *theta*, and a vanishing point *vp*, the line orientation voted for is $theta_{line}$ = arctan((*row* — *horizon*) / (*col* — *vp*)). The difference in angle between the line and the edge gradient estimate is $diff_{ang}$ = min(| $theta_{line}$ — *theta* |, 180 — | $theta_{line}$ — *theta* |). The vote for the line with vanishing point *vp* and orientation $theta_{line}$ has weight 1.0 — ($diff_{ang}$ / 90). In order not to bias the voting against lines near the corners of the image, the votes for a given line are normalized by the visible length of that line in the image.

The image is divided into a small number of horizontal bands (four to seven), and voting for the most popular vanishing point is done for each band. Once all the edge points have voted for all the lines they could lie on, the accumulator array is thresholded, and peaks in the accumulator array are detected. All the detected peak bins which support a given vanishing point are summed to give the total support for that vanishing point. The top three candidate vanishing points for each horizontal band in the image compete in a search for the globally best set of vanishing points. The criterion function for that search has a term for the strength of support for each vanishing point in the set, and a term which rewards continuity of features between adjacent bands in the image.

As a method for extracting road features, this technique has several advantages. The only calibration required is the determination of the horizon row, which can be easily computed from an image taken on a straight stretch of road. The model of road appearance used is fairly generic, assuming only that the road consists of features separated by constant widths and curving slowly enough that they can be approximated by straight lines within a horizontal band of the image. This eliminates the need to train the system separately for each road it encounters. It extracts as

much of the overall lane and road structure as it can detect, rather than just a right and left road edge or road centerline. If given a model of the road structure so that it could label the various features found, it could steer the vehicle down a specified lane using a pure pursuit strategy, once again without having to have any calibration from the image to the world other than a single gain parameter. While relatively slow in a serial implementation, the algorithm has a great deal of parallelism that could be exploited.

Figure 5.6 shows the results of the bootstrap algorithm on the image from figure 5.2. The algorithm successfully finds the white stripes on the left and right side of the road, the double yellow line in the middle, and part of the shoulder edge. There are a few extraneous edge segments caused by noise, for instance the tree shadow which runs parallel to the road.

This part of the research is still very experimental, and quantitative performance results are not yet available. Also, the issues of the best preprocessing to apply to the color image to produce the single-band image that the Sobel is run on, and how to set thresholds used in the algorithm are still under investigation.

So far we have described how YARF finds the location of the road in an initial image, how it locates individual features *given a prediction of the road location in subsequent images*, and how it combines those new data points into an updated estimate of the vehicle position on the road. Next we discuss our plans to integrate feature locations from multiple images into a local map, to reason about detected failures to locate features in order to detect changes in the road structure and the approach of intersections, and to use the integrated local map to navigate through intersections.

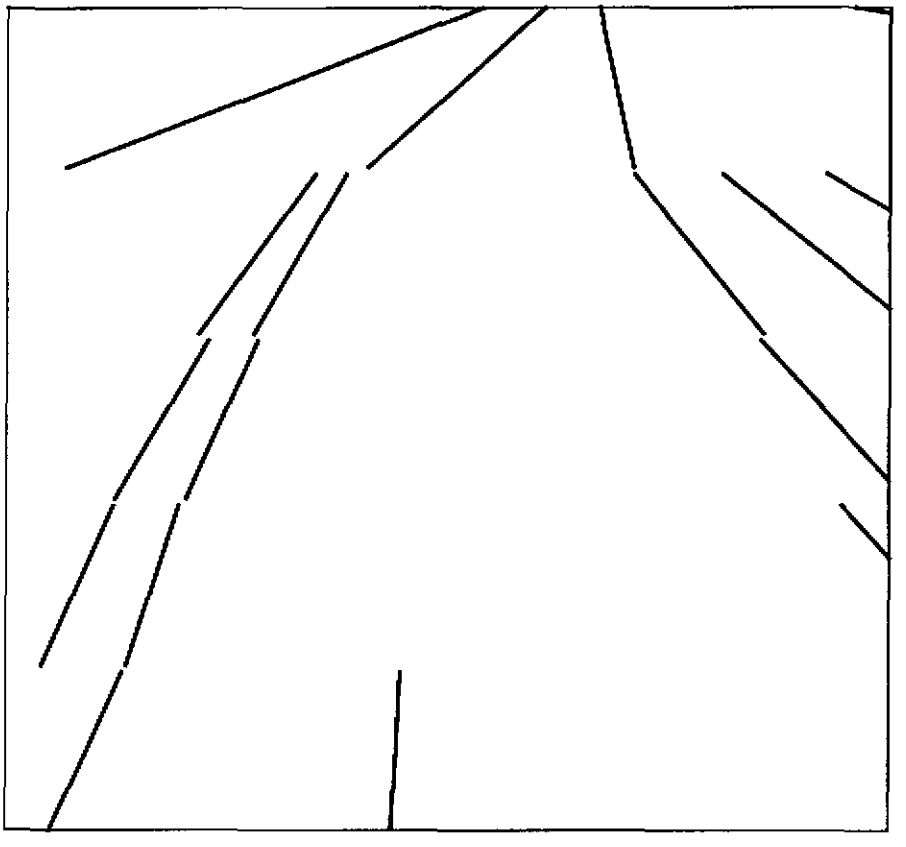


Figure 5.6: Line segments extracted by the vanishing point Hough algorithm

5.6 Intersection navigation

In order to navigate a vehicle through an intersection, an autonomous road following system must detect that the vehicle is approaching an intersection, use perception to locate the roads branching out from the intersection, and plan a path from the current lane through the intersection into the desired lane of the next road segment to be followed. More general intersection navigation capabilities than have been demonstrated in current systems require the coordination of perceptual data from multiple images into a single local map of the intersection.

There are several reasons for this. The first is that it gives the system a memory — once the system has detected an approaching intersection based on the disappearance of some of the road features, it does not have to devote processing cycles to remake this discovery on the following images. The second reason is that intersections of city streets cover a large area compared to typical camera fields of view. The integration of perception results from multiple images (both from the same camera over time as the vehicle moves, and from multiple cameras pointing in different directions to cover a larger field of view) is necessary in order to create a complete model of the intersection's geometry. Part of the NAVLAB project at CMU has been the creation of utilities to support an annotated map for robot navigation [19]. The function of the annotated map is to provide a framework for the communication of the results of different perception modules through a shared geometric database. YARF will use the annotated map facility to store results from multiple images in a common coordinate system.

Our first experiments will focus on the question of whether the results from multiple images taken by multiple cameras can be combined to produce a coherent, accurate map of the scene geometry. The coherence of data from multiple frames taken by a moving camera (see figure 5.5) is promising. In our initial experiments we will run the road following process with two different cameras at the same time and examine how well the feature positions match between the two cameras.

The next step is to use local map data to detect the approach of an intersection. The annotated map will be used to store information about locations where features were not detected where they were expected. YARF will then reason about the missing feature data to determine whether the road model has changed, whether a feature has become obscured, whether a different segmentation technique should be switched to because of a change in feature appearance, or whether the vehicle is approaching an intersection.

After YARF has the capability to detect that the vehicle is approaching an intersection, the final step is to create perception strategies for locating the roads branching out of the intersection. Rather than assume complete knowledge of the intersection geometry as the Sidewalk II system did, YARF will assume only knowledge of the feature cross-sections of the roads which meet at the intersection. The current plan is to use a feed-forward tracking strategy similar to that used by MARF to follow road features around corners and through the intersection, using the feature cross-section models to predict feature locations once an initial estimate of a road branch's location is available. YARF will use multiple cameras to cover a wider field of view, using calibration information to track features across the overlapping fields of view.

5.7 Conclusion

The YARF project has made substantial progress since we reported our first results. We have gone from an initial "pot-luck" collection of segmentation techniques to focused research into robust techniques to track different types of road feature. We have implemented routines to fit individual estimates of road feature locations to models of generalized stripe roads whose spines are locally approximately circular arcs, and are investigating issues of filtering and the use of robust estimation to improve reliability. We have made preliminary experiments in the initial location of road features using Hough line detection techniques and shared vanishing point constraints. We have a plan of research to add intersection navigation capabilities into the system. YARF has driven the NAVLAB at

speeds of up to 6.75 MPH on a public road running through a golf course near campus, and we expect speed improvements from the use of multiple processors.

Other research within the NAVLAB project at CMU has focused on planning in the domain of driving in traffic on city streets, using a simulator (PHAROS) to provide the input of the system [16]. As the YARF project progresses, it will provide some of the perceptual capabilities needed to transfer results from the research using the PHAROS simulator into the real world. It provides an open ended architecture which improved segmentation techniques can easily be plugged into to improve system performance.

5.8 Acknowledgements

Our thanks to Thad Druffel, who worked on various pieces of the system, and implemented a multiprocessor version of YARF.

This research is sponsored by DARPA, DOD, monitored by the US Army Engineer Topographic Laboratories under contract DACA 76-89-C-0014, titled "Perception for Outdoor Navigation".

5.9 References

[1] Aubert, Didier, and Thorpe, Chuck.
Color Image Processing for Navigation: Two Road Trackers.
Technical Report CMU-RI-TR-90-09, Robotics Institute, Carnegie Mellon, April, 1990.

[2] Besl, Paul J., Birch, Jeffrey B., and Watson, Layne T.
Robust Window Operators.
In *Proceedings International Conference on Computer Vision.* 1988.

[3] Binford, T. O.
Survey of model-based image analysis systems.
Int. J. Robotics Research 1, 1981.

[4] Crisman, Jill.
Color Vision for the Detection of Unstructured Roads and Intersections.
PhD thesis, Carnegie-Mellon University, 1990.

[5] DeMenthon, Daniel, and Davis, Larry.
Reconstruction of a Road by Local Image Matches and Global 3D Optimization.
In *Proceedings 1990 IEEE International Conference on Robotics and Automation.* May, 1990.

[6] Dickinson, S., and Davis, L.
An Expert Vision System for Autonomous Land Vehicle Road Following.
In *Computer Vision and Pattern Recognition Conference.* 1988.

[7] Goto, Y., Matsuzaki, K., Kweon, I., and Obatake, T.
CMU Sidewalk Navigation System: A Blackboard-Based Outdoor Navigation System Using Sensor Fusion with Colored-Range Images.
In *Proc. Fall Joint Computer Conference.* November, 1986.

[8] Kenue, Surender K.
LANELOK: Detection of Lane boundaries and Vehicle Tracking Using Image-Processing Techniques -- Part I: Hough-Transform, Region Tracing and Correlation Algorithms.
In *SPIE Mobile Robots IV.* 1989.

[9] Kenue, Surender K.
LANELOK: Detection of Lane boundaries and Vehicle Tracking Using Image-Processing Techniques -- Part II: Template Matching Algorithms.
In *SPIE Mobile Robots IV.* 1989.

[10] Kluge, Karl, and Thorpe, Charles E.
Explicit Models for Robot Road Following.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 3.

[11] Kuan, Darwin; Phipps, Gary; and Hsueh, A.-Chuan.
Autonomous Land Vehicle Road Following.
In *Proceedings First International Conference on Computer Vision.* June, 1987.

[12] McKeown, David M., and Denlinger, Jerry L.
Cooperative Methods for Road Tracking In Aerial Imagery.
In *Proceedings Computer Vision and Pattern Recognition.* June, 1988.

[13] Mysliwetz, Birger D., and Dickmanns, E. D.
Distributed Scene Analysis for Autonomous Road Vehicle Guidance.
In *Proceedings SPIE Conference on Mobile Robots.* November, 1987.

[14] Polk, Amy, and Jain, Ramesh.
A Parallel Architecture for Curvature-Based Road Scene Classification.
In *Roundtable Discussion on Vision-Based Vehicle Guidance '90 (in conjunction with IROS).* July, 1990.

[15] Pomerleau, Dean A.
Neural Network Based Autonomous Navigation.
Vision and Navigation: The Carnegie Mellon Navlab.
Kluwer Academic Publishers, 1990, Chapter 5.

[16] Reece, Douglas A., and Shafer, Steve.
An Overview of the PHAROS Traffic Simulator.
In *Proceedings of the Second International Conference on Road Safety.* September, 1987.

[17] Schaaser, L. T., and Thomas, B. T.
Finding Road Lane Boundaries for Vision Guided Vehicle Navigation.
In *Roundtable Discussion on Vision-Based Vehicle Guidance '90 (in conjunction with IROS).* July, 1990.

[18] Thorpe, Charles; Hebert, Martial; Kanade, Takeo; and Shafer, Steven A.
Vision and Navigation for the Carnegie-Mellon Navlab.
IEEE Transactions on Pattern Analysis and Machine Intelligence 10(3), May, 1988.

[19] Thorpe, Charles, and Gowdy, Jay.
Annotated Maps for Autonomous Land Vehicles.
In *Proceedings of the DARPA Image Understanding Workshop.* 1990.

[20] Turk, Matthew A.; Morgenthaler, David G.; Gremban, Keith D.; and Marra, Martin.
VITS -- A Vision System for Autonomous Land Vehicle Navigation.
IEEE Transactions on Pattern Analysis and Machine Intelligence 10(3), May, 1988.

[21] U. S. Bureau of the Census.
Statistical Abstract of the United States: 1988.
U. S. Bureau of the Census, 1987.

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

[23] A. Waxman, J. LeMoigne, L. Davis, B. Srinivasan, T. Kushner, E. Liang and T. Siddalingaiah.
A visual navigation system for autonomous land vehicles.
Journal of Robotics and Automation, Vol. 3 , 1987.

List of Figures

ANNOTATED MAPS

WARP ON NAVLAB

YARF

List of Figures