June 1987 Annual Report: Development of an Integrated Mobile Robot System at Carnegie Mellon

Steve Shafer and William Whittaker
Principal Investigators

CMU-RI-TR-88-10

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

July 1987

This research is sponsored by the Defense Advanced Research projects Agency, DoD, through DARPA order 5682, and monitored by the U.S. Army Engineer Topographic Laboratories under contract DACA76-86-C-0019. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the **official** policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government.

Table of Contents

Abstract 1

I: Introduction *Steve Shafer and Red Whittaker* 3
- Introduction 3
- Accomplishments 5
- Personnel 6
- Publications 6

II: NAVLAB: An Autonomous Navigation Testbed 7
K. Dowling, R. Guzikowski, J. Ladd, H. Pangels, J. Singh, and W. Whittaker

III: Vision and Navigation for the Carnegie-Mellon NAVLAB 53
C. Thorpe, M. Hebert, T. Kanade, and S. Shafer

IV: The CMU System for Mobile Robot Navigation 81
Y. Goto and A. Stentz

Abstract

This report describes progress in development of an integrated mobile robot system at the Carnegie Mellon Robotics Institute from July 1986 to June 1987. This research was sponsored by DARPA as part of the Strategic Computing Vision program.

Our program includes a broad agenda of research in the development of mobile robot vehicles. In the year covered by this report, we addressed two major areas in vehicle development (*NAVLAB vehicle* and *Robot control system*) and two major areas in robot architecture development (*CODGER blackboard* and *Navigation architecture*):

1. **NAVLAB vehicle.** We built the NAVLAB mobile robot vehicle by outfitting a commercial truck chassis with computer-controlled drive and steering controls and a set of on-board computer workstations. The NAVLAB serves as a mobile navigation laboratory that allows researchers to interact intensively with the system during testing and execution.
2. **Robot control system.** We developed a real-time controller system for the NAVLAB using a collection of coordinated processors and software. This system implements a Virtual Vehicle that can interpret and execute commands from the high-level planning system. The Virtual Vehicle hides many of the physical details from the higher level system so that research in perception and planning can take place very rapidly.
3. **CODGER blackboard.** We designed and implemented the CODGER blackboard system for robot perception and reasoning on a distributed collection of processors. CODGER incorporates substantial features for geometric reasoning and task synchronization that have not been incorporated in blackboards before. These features are essential for robot navigation.
4. **Navigation architecture.** We developed the Driving Pipeline architecture for coordinating road following, obstacle avoidance, and vehicle motion control. The Driving Pipeline allows these functions to be combined in a single system and provides continuous motion rather than stop-and-go.

This hardware and software combination is the basis for the **New Generation System** (NGS) for robot vision and navigation, which will tie together existing and emerging technologies

This report begins with an introduction, summary of achievements, and lists of personnel and publications. It also includes relevant papers resulting from this research.

Section I

Introduction

Introduction and Overview

This report reviews progress at Carnegie Mellon from July 1, 1986, to June 30, 1987, on research sponsored by the Strategic Computing Initiative of DARPA, DoD, through ARPA Order 5682, and monitored by the US Army Engineer Topographic Laboratories under contract DACA76-85-C-0019, titled "Development of an Integrated Mobile Robot System." This report consists of an introduction and overview, and detailed reports on specific areas of research.

During this time period we designed and built a robot vehicle, the NAVLAB, as a tool and testbed for research in robot navigation, and we developed a software framework for integrating vision, planning, and control modules into a single working system. The modules themselves were developed under a related research effort in "Road Following", which is also sponsored by DARPA. The total system has been demonstrated in outdoor navigation runs without human intervention, on a road in Schenley Park, Pittsburgh, near the Carnegie Mellon campus. The vehicle and architecture developments are described briefly below, and at length in the following chapters of this report.

Development of the NAVLAB Mobile Robot Vehicle. The hardware portion of this research was aimed at accomplishing two tasks: *developing* a mobile robot research testbed vehicle, and *outfitting* it with the necessary controllers, sensors, and computers to support increasingly ambitious navigation system demonstrations. To fulfil these tasks, we developed the NAVLAB mobile robot vehicle to serve as a testbed for research in outdoor navigation, image understanding, and the role of human interaction with intelligent systems.

Vehicle development: The NAVLAB is a roadworthy truck modified so that humans or computers can control it as the occasion demands. It acts as a mobile navigation research environment, accommodating onboard researchers as well as significant onboard computing power. Because it is self-contained, it is not subject to telemetry bottlenecks, communication faults, or dependence on stationary infrastructure. The NAVLAB can travel to confront navigation problems of interest at any site.

The NAVLAB locomotion configuration consists of a chassis, drivetrain, and shell. The chassis is a modified, cut-away van with a computer-controllable hydraulic drivetrain. Driver's controls allow a human monitor to override automatic control for overland travel, position the vehicle for each experimental run, and recover from errors as the software is improved and debugged. The shell houses all onboard equipment including computers, controllers, telemetry, and internal sensors. In addition, it provides a working area for operators, allowing researchers to be in intimate contact with the computer programs that are controlling the vehicle. Equipment racks, seating, lighting, air-conditioning, desk space, monitors, and keyboards create a mobile environment for research.

Outfitting of the vehicle: The core of the NAVLAB is the vehicle controller. In the autonomous mode, this multi-processor computer controls all locomotion, actuation, and physical sensing. It interacts with a computer host and human operator to provide varying degrees of autonomy. The NAVLAB controller queues and executes "Virtual Vehicle" commands originating from a computer or human host. This command set provides high-level motion and control primitives that mask the physical details of the vehicle, and is extensible for control of other mobile systems.

The NAVLAB also supports a choice of sensing equipment to accomodate many types of navigation perception research. Video cameras provide color and intensity images for scene interpretation. A scanning rangefinder is used to sweep the surroundings with a distance-measuring laser that provides three-dimensional information about the geometry and reflectivity of the environment. Taken together, data concerning color, intensity, range, and reflectance provide a rich basis for building scene descriptions in the complex outdoor environment. The sensor information from these sources is fused together to achieve robust perception.

Finally, the NAVLAB incorporates a network of several general-purpose (SUN) computer workstations, complete with operators' keyboards and monitors, connected by an on-board Ethernet. These workstations perform the perception and planning tasks for the NAVLAB, and communicate with the sensors and the Virtual Vehicle controller to receive sensor data and issue commands for vehicle motion. The use of general-purpose computers in the NAVLAB makes available a full range of software development tools for use by researchers in perception, path planning, and navigation.

Both the vehicle development and its outfitting with sensors and computers are described in the following report, "NAVLAB: An Autonomous Navigation Testbed".

Development of an Integrated Software System for Navigation. The software integration effort is designed to produce a single system that incorporates all of the key technologies needed for an autonomous vehicle to perform sophisticated missions. In the past, efforts to build outdoor mobile robot vehicles have fallen into one of two categories, according to the mission they were designed to address: *road-following* systems that use the road edges for navigation, with minimal ability to detect and avoid obstacles; and *cross-country* systems that use range-finders to traverse open country, with little or no ability to cross or travel on roads. Our research is aimed at developing the first robot vehicle to incorporate road-following, cross-country travel, obstacle detection and avoidance, object recognition, and mission planning all in one single framework. In order to achieve this, substantial progress is needed in all of these individual technologies as well as the building of the complete system. The system-building task is the goal of the Software portion of this research. There are two parts to this task: development of a *software framework* for system integration, and design of a *navigation architecture* to coordinate the various vision, planning, and control modules of the system. Together, these form the backbone of the New Generation System (NGS) for Vision.

Software framework for system integration: The CODGER blackboard. We have developed the CODGER blackboard for robot navigation, which incorporates several features needed for mobile robot system development. The structure of CODGER is a central database to which any number of modules can be attached. The whole system runs on a collection of general-purpose machines (SUNs and VAXes, with Ethernet connection), giving the module programmers the best possible software development environment. CODGER implements a central database that is utilized by the modules acting in concert as a distributed system. Each module is a self-contained program that stores data in CODGER and can retrieve data. Much of the data is of a standard form long known in database design: strings, numbers, arrays, etc. However, to support robot planning and perception, a tremendous amount of *geometric* computation is also needed. For that reason, a complete set of geometric primitives and operations is also included in the CODGER system. An example of its use is this: a road-following module finds a road polygon on the ground, and an obstacle detection module locates objects to be avoided; the path-planning module can then ask CODGER to find the set of obstacles that lie within the detected road area. In addition to these basic geometric operations, CODGER has a special mechanism for managing multiple coordinate frames. This kind of geometric reasoning has not been incorporated within a general-purpose database in the past, and it is essential for robot navigation.

Navigation architecture: The Driving Pipeline In addition to the CODGER blackboard, the software integration effort has produced a design for a collection of modules to utilize this blackboard and implement initial versions of the integrated NGS. This design has been implemented and demonstrated in road-following tasks on several occasions through 1986 and early 1987.

The Driving Pipeline is one of the key contributions of the software integration effort, and introduces two new concepts for robot vehicle system design. The first of these is the *driving unit*, a "chunk" of roadway or off-road terrain that can be seen in a single view of the camera or range-finder. This unit will be predicted from the map, utilized for both kinds of perception, then passed along to the path planner for planning. Finally, the resulting path is passed on to the vehicle control system. The use of driving units allows all activities to be coordinated easily, since they all process the same stretch of roadway in turn. The entire length of the route followed by the vehicle is broken into driving units as the vehicle travels. Driving units are also a convenient way to integrate intersections (each is one driving unit) and cross-country terrain (broken into small patches) into the total system.

If each driving unit had to be processed from start to finish as the vehicle passes from one to the next, it would require that the vehicle stop at the end of each driving unit, process the next one, then begin moving. In order to provide a higher rate of processing and allow continuous motion, the software modules are arranged into a *Driving Pipeline* that uses the concept of pipelined processing to provide a higher throughput. In the Driving Pipeline, the prediction is made three units ahead of the vehicle's current position. Perception takes place by looking ahead two units in advance of the vehicle, where the prediction was already completed; path planning, which must follow perception, occurs on the unit directly in front of the vehicle. Finally, the vehicle control operates on the driving unit the vehicle is currently traversing. With the Driving Pipeline, the various processing steps can proceed in parallel on successive driving units in front of the vehicle. The NAVLAB proceeds using continuous motion by means of the streamlining provided by the Driving Pipeline.

These software developments -- the CODGER blackboard and the Driving Pipeline architecture -- are described in more detail in the report below, "The CMU System for Mobile Robot Navigation." The entire system, including CODGER, the Driving Pipeline, and the perception processes, is described in the following report, "Vision and Navigation for the Carnegie-Mellon NAVLAB."

Accomplishments

The key accomplishments of this research in the time period from July 1986 to June 1987 have been:

- Development of computer controls for the NAVLAB vehicle.
- Development of real-time control system for NAVLAB.

- Implementation of Virtual Vehicle command interface for the control system.
- Development of CODGER blackboard system for geometric reasoning and synchronizing distributed tasks.
- Integration of subsystems for road following, obstacle avoidance, and path planning.
- Design of Driving Pipeline architecture for continuous motion.
- Implementation and demonstration of complete system in Schenley Park, Pittsburgh.

Personnel

Directly supported by the project, or doing related and contributing research:

Faculty: Takeo Kanade, Steve Shafer, Chuck Thorpe, William Whittaker.

Staff: Mike Blackwell, Tom Chen, Jill Crisman, Kevin Dowling, Ralph Hyre, Jim Ladd, Jim Martin, Jim Moody, Tom Palmeri, Jeff Singh, Chuck Whittaker, Eddie Wyatt.

Visiting scientists: Yoshi Goto, Taka Fujimori.

Graduate students: Rob Guzikowski, Joe Kuefler, InSo Kweon, Henning Pangels, Doug Reece, Tony Stentz.

Publications

Selected publications by members of our research group, supported by or directly related to this contract:

1. *NAVLAB: An Autonomous Navigation Testbed.* Dowling, K., et al. Technical report CMU-RI-TR-87-24, November 1987.
2. *Mobile Robot Navigation: The CMU System.* Goto, Y. and Stentz, A. *IEEE Expert*, Winter 1987, p.44-54.
3. *The CMU System for Mobile Robot Navigation.* Goto, Y. and Stentz, A. Presented at IEEE Intl. Conference on Robotics and Automation, April 1987, p.99-105.
4. *A Feasibility Study for a Long Range Autonomous Underwater Vehicle.* Hebert, M., et al. In *Proc. Fifth Intl. Symposium on Unmanned Untethered Submersible Technology*, June 1987, p.1-13.
5. *Error Modeling in Stereo Navigation.* Matthies, L. and Shafer, S. *IEEE J. Robotics and Automation*, June 1987, p.239-248.
6. *Vision and Navigation for the Carnegie-Mellon NAVLAB.* Thorpe, C. et al. *Annual Reviews of Computer Science* 1987, vol. 2, p.521-556.
7. *1986 Year End Report for Road Following at Carnegie Mellon.* Thorpe, C. and Kanade, T. Technical Report CMU-RI-TR-87-11, May 1987.

Section II

NAVLAB: An Autonomous Navigation Testbed

Kevin Dowling
Rob Guzikowski
Jim Ladd
Henning Pangels
Jeff Singh
William Whittaker

Abstract

The NavLab is a testbed for research in outdoor navigation, image understanding, and the role of human interaction with intelligent systems. It accommodates researchers and all computing onboard. The core of the NavLab is the vehicle controller, a multi-processor computer that controls all locomotion, actuation and physical sensing; it interacts with a computer host and human operator to implement varying degrees of autonomy. The chassis is a modified van with a computer-controllable, hydraulic drivetrain. The NavLab supports a choice of sensing to accommodate many types of navigation research. This report details the control computing and physical configuration of the NavLab vehicle.

1. Introduction

The NavLab is a testbed for research in outdoor navigation, image understanding, and the role of human interaction with intelligent systems. A mobile navigation habitat, it accommodates researchers and significant onboard computing. Applications for field navigation vehicles include mapping of hazardous waste sites, off-road haulage, material handling at construction worksites, and exploration of planetary surfaces.

The NavLab is a roadworthy truck modified so that humans or computers can control as occasion demands. Because it is self-contained, it is not subject to telemetry bottlenecks, communication faults or dependence on stationary infrastructure, and can travel to confront navigation problems of interest at any site.

The core of the NavLab is the vehicle controller. In autonomous mode, this multi-processor computer controls all locomotion, actuation and physical sensing. It interacts with a computer host and human operator to implement varying degrees of autonomy. The NavLab controller queues and executes Virtual Vehicle commands originating from a computer or human host. This command set provides high-level motion and control primitives that mask the physical details of the vehicle, and is extensible for control of other mobile systems.

The NavLab configuration consists of a chassis, drivetrain and shell. The chassis is a modified, cut-away van with a computer-controllable, hydraulic drivetrain. Driver's controls allow a human monitor to override automatic control for overland travel, setup and recovery from experimental errors. The shell houses all onboard equipment including computers, controllers, telemetry, and internal sensors. In addition, it provides a working area for operators, allowing research within the confines of the vehicle. Equipment racks, monitors, lighting, air-conditioning, seating and desk space create a mobile environment for research.

Humans can monitor and supervise the NavLab from the operator's console for setup, error recovery and tuning. Interface modes include Virtual Vehicle instructions, joystick motion control, and direct servo motion commands. The console also incorporates several displays to show the current states of both the vehicle and control computer.

The NavLab supports a choice of sensing to accommodate many types of navigation research. Video cameras provide color and intensity images for scene interpretation. NavLab vision experiments use a single camera to analyze road edges through intensity, texture, and color segmentation. A scanning rangefinder sweeps the surroundings with a distance-measuring laser that provides useful three-dimensional information about the geometry and reflectivity of the environment. Laser experiments navigate through geometric features like trees and buildings. Taken together, data of color, intensity, range and reflectance provide a rich basis for building natural scene descriptions. Sensor information from several sources can be fused to achieve more robust perception. A blackboard architecture integrates the distributed processes that sense, map, plan and drive.

The NavLab represents continuing evolution in the design of navigation vehicles. Fully self-contained, it is a milestone in mobile robotics.

This technical report details the control computing and physical configuration of the NavLab vehicle. Information on other aspects of the NavLab, including perception, modelling, planning and blackboard architectures, can be found in articles listed in Appendix V.

2. Controller

The NavLab controller parses and implements a Virtual Vehicle instruction set. The controller is implemented as a loosely coupled parallel processor. Commands are received via a serial link from Host computers or an onboard console. Five axes of motion are controlled: drive, steering, pan and tilt motions for the cameras, and a pan motion for a laser ranging device. Status of devices onboard is monitored by a sensor subsystem that constantly polls processors dedicated to groups of sensors via a high-speed serial bus. Status information is displayed on the console inside the vehicle and is available to the Host computer via queries.

2.1 System Architecture

The control computing for the NavLab is based on the hierarchy shown in Figure 2-1, a system architecture for robot modeling and planning associated with autonomous task execution in unstructured and dynamic environments. The NavLab controller is tantamount to the lowest level of the architecture. The need for an interface protocol between the control computing and the higher level computing forges the virtual system, which allows the low-level control computing to mask the physical implementation details from the higher level computing. This is accomplished through command primitives that define the interface. Using the virtual system, many of the high-level modeling and planning functions can port across a number of different physical systems that can be controlled with the same command primitives. Only the lowest level control computing must deal with the physical differences of the system.

Complex systems usually defy any attempts at mathematical modeling techniques, which makes control parameters impossible to even estimate. A set of pseudo constants, tunable from an operator's console, adjust the parameters and gains. The NavLab maintains these constants in file structures that remain on disk for power-up initialization. The system always starts up with default values established from the most recent tuning of the system.

The control computing accepts commands from a host or human operator who can intervene at various levels of control to insure safe operation during experiments. The assumption during development is that the higher levels of computing will not succeed in all situations. The control computing thus provides a graceful transition between computer and human control when failures occur. The hooks for human inputs are also useful for setup and recovery during experiments.

The sensors monitored by control computing reflect the state and ability of the system to respond to commands issued by the cognitive planning layers. The values of the observed parameters have fixed maximum limits that are characterized by the physical system. These limits, however, are not static and can move inward during certain operating conditions. Physical parameters such as heat and pressure can diminish NavLab's mechanical ability to respond to commands. The parameter limits are dynamically adjustable by the controller to protect the NavLab. When controlling with a powerful physical plant like the NavLab, erroneous plans and commands have significant impact. The control computer should never execute commands blindly, so mechanisms are needed for validating and rejecting commands, with advisories communicated to the source of commands.

Control computing, the lowest of the three levels within the autonomous mobile system architecture, interacts with the physical system. The design criteria set forth for the NavLab low-level controller include an open-ended architecture, a virtual system interface and multiple command streams.

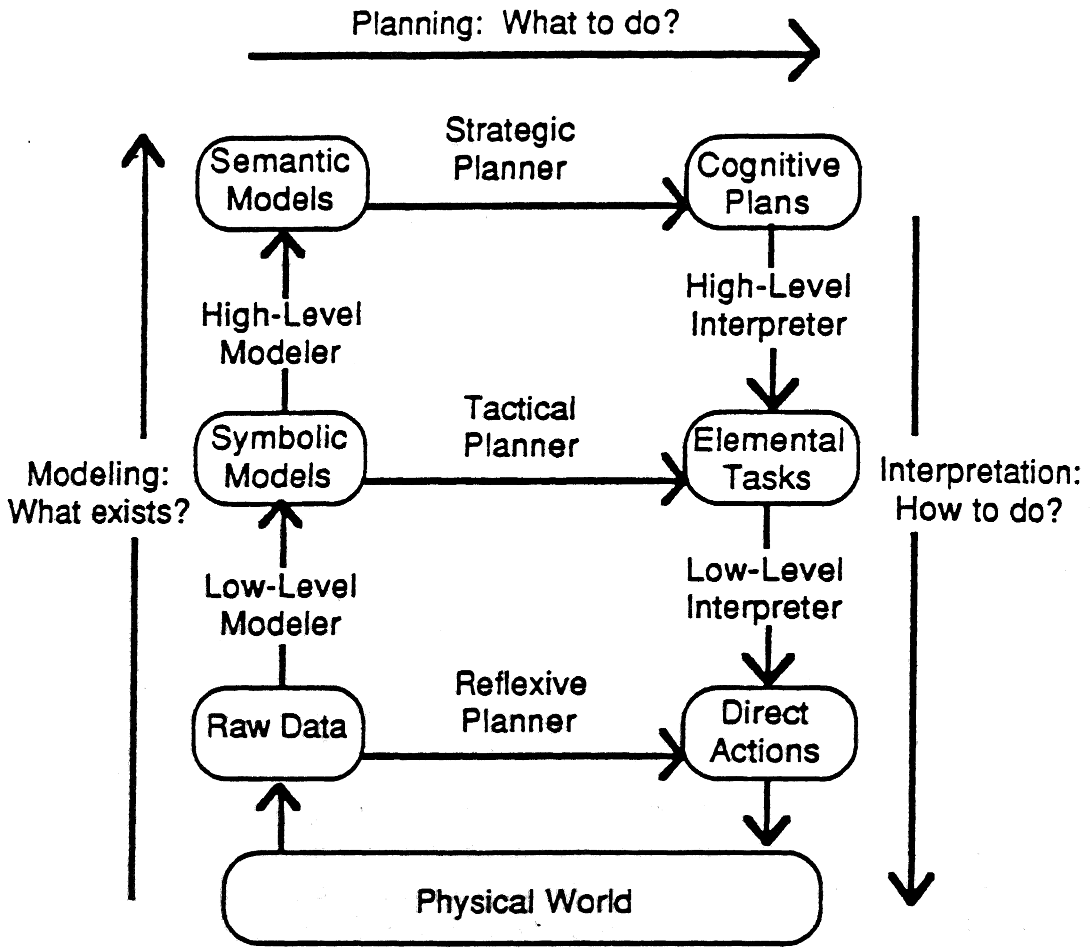


Figure 2-1: The Hierarchical Layering of a System Architecture for Modeling and Planning

2.2 Virtual Vehicle

A Virtual Vehicle is a man-machine interface that accepts conceptual commands and provides a clean separation between the navigation host and vehicle control. This interface masks implementation details of the physical vehicle, facilitating adaptability to future navigation testbeds.

The Host (the computing engine that does planning) communicates with the Virtual Vehicle via ASCII data transmitted over a serial line. The communication falls into three categories:

- *Commands* issued by the Host
- *Queries* by the Host about the status of devices
- *Reports* initiated by the Virtual Vehicle on completion of commands and in case of emergency.

In the current implementation, the vehicle is directed along circular arcs because arcs are quickly computed and absolute position is not critical (the arcs are being upgraded to clothoids). Because it is not possible for a vehicle to switch between an arc of one curvature and another instantly, path transitions

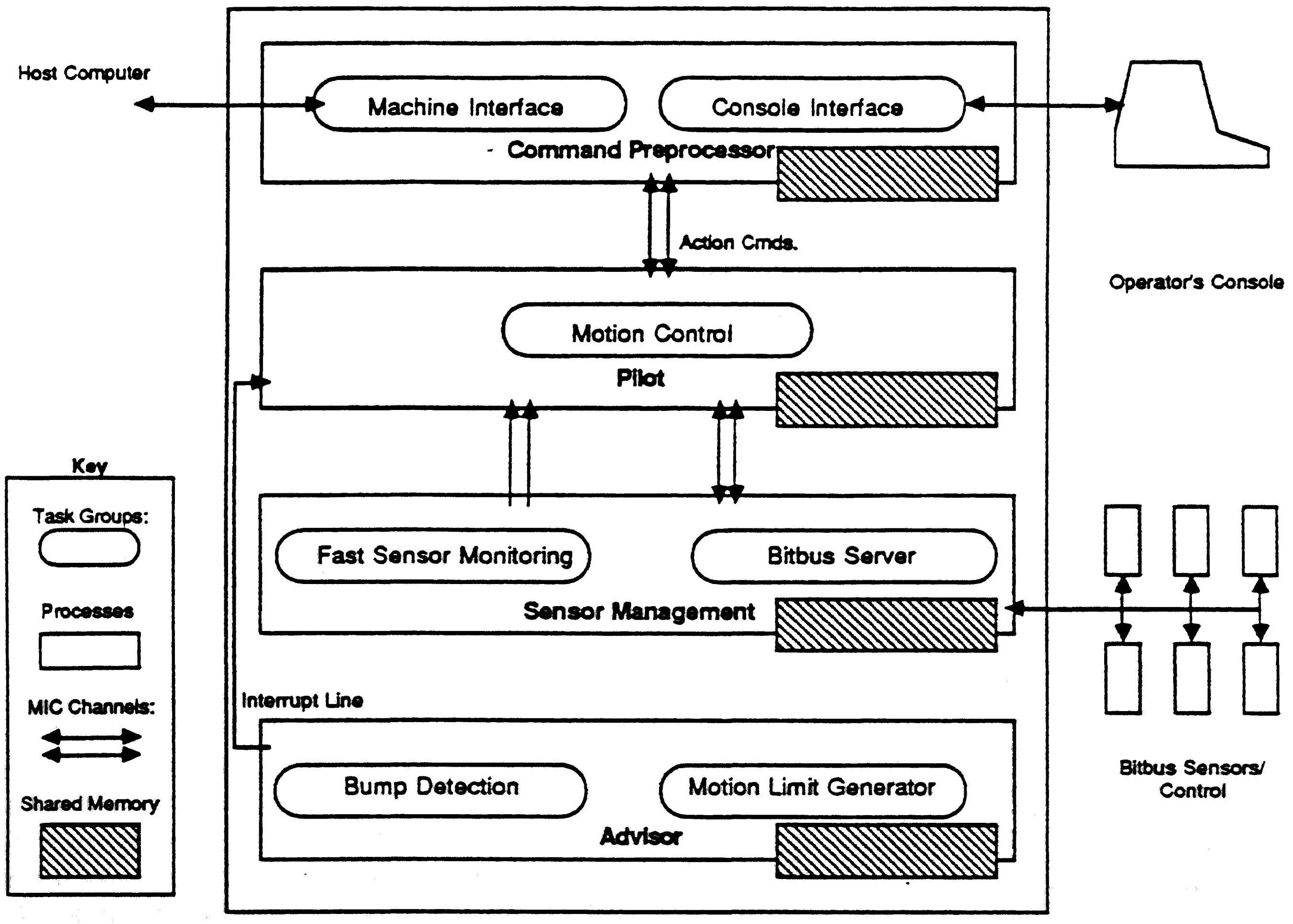


Figure 2-2: Architecture of Controller

boards and shared memory spaces. Critical memory and I/O locations are controlled using a semaphore system while bus contention is arbitrated in hardware. Interrupt lines in the Multibus backplane tie the processors together for inter-processor communications.

Each processor is identically configured with 256 K local ROM, 512 K local RAM, and a 256 K window to the Multibus. The ROMs on the I/O processors only contain operating system software and a download facility to allow loading of applications. The multiple bus structure permits a total system memory of 2.5 MB even though only 1 MB is addressable from each processor.

The controller also contains intelligent slave boards for I/O expansion and servo motor control. These boards may be accessed by any bus master. Often, access is restricted to a specific processor to avoid contention problems.

are inexact. Errors are compensated for by dynamically planning arcs to reach subgoal points along the path.

To facilitate synchronization, all drive and steering commands are initiated at the transitions between arcs. The capability is provided to make changes to vehicle motion (e.g., curvature of the arc, vehicle velocity) on the fly. Arcs (specified as [arclength, radius of curvature]) can be queued for sequential implementation.

The Virtual Vehicle and Host interact as follows:

- The Host issues a new arc command before the arc in execution is completed.
- If an immediate condition is specified, the old arc is discarded and the new arc is accepted immediately. Otherwise, the new arc is initiated at the end of the arc being executed.
- When a new arc is initiated, vehicle position is reported to the Host for use in calculating future path plans.
- The Host incorporates the reported position in planning the next arc, thereby compensating for deviations from the desired path.

The Virtual Vehicle instruction set and details of interfacing can be found in Appendix IV.

2.3 Controller Architecture

The NavLab controller is a powerful and flexible multi-processor system. A functional block diagram of the controller is shown in Figure 2-2. A Pilot module, responsible for management and operation of the key peripherals and I/O devices in the system, maintains direct control of all physical action and motion. The Pilot is also responsible for system startup and synchronization and acts as the hub in a star configuration for inter-processor communication. A Command Preprocessor manages I/O between the controller and devices that communicate with it. The Sensor Manager controls a network of 8-bit micro-controllers distributed throughout the vehicle to provide points of intelligent analog and digital I/O. Accommodations are made for an Advisor to set limits on physical motion parameters based on the perceived condition of the mechanical systems of the vehicle. The Advisor incorporates a bump detection subsystem that signals the Pilot if immediate action is necessary.

Each module in the system contains its own operating environment for independent/parallel operation. The operating environments are subsets of those used for system development. Code for each module is down-loadable to permit easy modification to the system.

2*3.1 Hardware

The WavLato controller is designed as a two-tiered multi-processor system. The first tier is responsible for the primary computing, control I/O and motion control. It is comprised of 4 Intel 28612 processor boards residing in a common Multibus backplane. The second tier performs remote data acquisition and control of devices located around the vehicle using a serial network of 3-bit micro-controllers. The Sensor Management System in the first tier is the interface between the two tiers.

2.3*1.1 Primary Computing

Processors in the first Her take advantage of the multiple bus structure of the system to increase processing throughput Each processor contains a local bus with enough memory resources to support is own execution environment Processors have bus master capabilities to access and control I/O

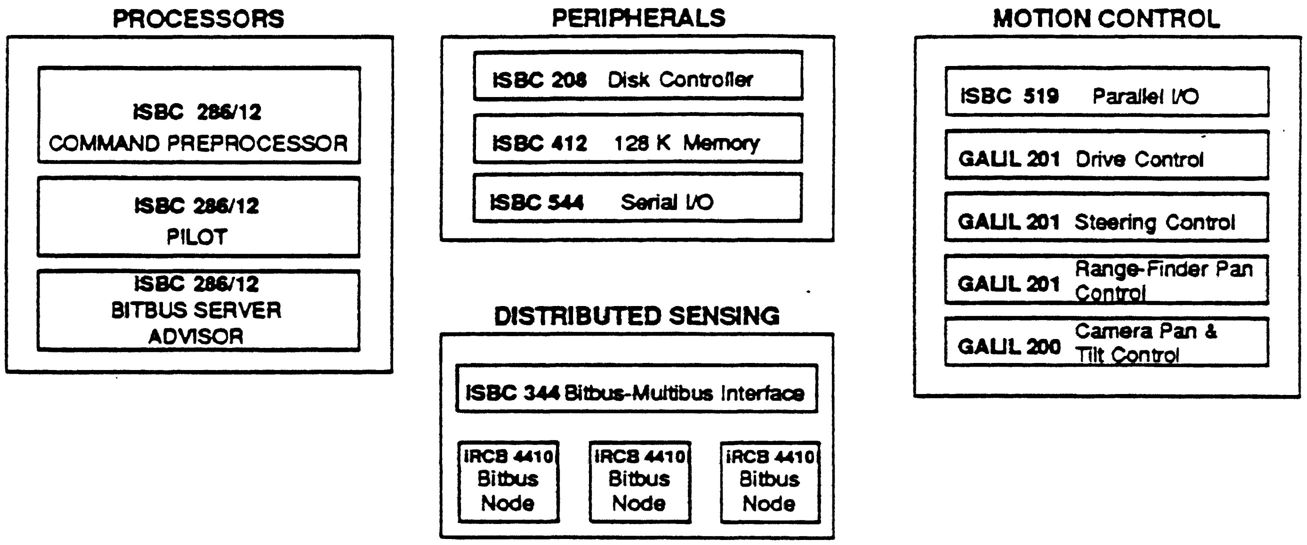


Figure 2-3: Hardware Configuration

2.3.1.2 Secondary Computing

The second computing tier physically distributes sensing and control of devices around the vehicle by using nodes that consist of 8-bit microcontrollers communicating over a high-speed serial bus using a message protocol called Bitbus. The network is controlled by a Master node that either continually polls other nodes to read analog and digital inputs or continuously commands them with reference values. The communication overhead makes Bitbus suitable to tasks that require high-level control and slow data acquisition. The serial bus network is extensible to support up to 250 nodes. Further implementation details of Bitbus can be found in Section 2.3.5.1.

2.3.2 System Software

System software for the controller is based on the iRMX 86 real-time operating system. iRMX is configurable to customize operating environments for each processor in the system. These operating environments are resident in ROM local to each board and are booted from reset. iRMX 86 provides objects to support an event-driven, multi-tasking environment.

A facility to down-load object code was developed for building and testing multiprocessor systems. A single processor accesses the mass storage device containing object code files for downloading. This processor, like a system server, loads object code into shared memory and signals the appropriate target board when a valid record is available. The other processors contain consumer jobs to copy records from shared memory to their local memory. On signal from the server processor, a consumer job releases the server CPU to allow the local Operating System to start the jobs from the newly loaded code. Once the application software is running, the consumer load job lies idle and waits for a signal from the server processor to reset and begin the load sequence again. This flexible load facility is a valuable tool for building and testing multiprocessor systems.

2.3.2.1 Interprocessor Communication

Processors communicate using shared memory in two different ways. Common variables are accessed by multiple processors to share state information (scratch pad communication). Messages can also be written to specific memory locations on other processor boards and the receiving board is signaled by an interrupt. This method is often used by one board to direct processing on another board.

Scratch Pad Communication

This method is a simple solution to sharing a large amount of data between modules. Processes that acquire data (status of devices, vehicle orientation, speed, etc.) post this information to the scratch pad area instead of sending the data to all modules that need them. Most shared variables are independent of each other; hence contention problems are limited to access of the individual memory locations to read or write. Dependent variables (ones that must be accessed as a group) require a software semaphore to provide mutual exclusion. An indivisible test and set instruction provided by PL/M-86 was used to create the semaphore system.

Module for Interprocessor Communications

The Module for Interprocessor Communication (MIC) was developed to support flexible pipelined communications between tasks running on separate processor boards. MIC provides the applications programmer with a simple set of procedure calls from which a task can queue messages containing a board and task destination. MIC handles the transfer of these messages between boards.

MIC is implemented as a star architecture. All messages are sent through a central node to limit the number of required interrupt lines. This scheme is well suited to the NavLab controller because most interprocessor communications are to a central node (the Pilot).

MIC was built using tools provided by iRMX including inter-task communication, dynamic memory allocation, and FIFO queues. MIC runs as an interrupt-driven task. It responds to signal interrupts to determine the destination of a received message and then sends it to the appropriate task. iRMX system calls permit asynchronous message transfers between tasks.

MIC was designed to be compact (5 K), fast, and capable. MIC is able to dynamically allocate message segments to meet the load of interprocessor communication traffic that varies from processor to processor. This prevents wasting memory and time required by the system programmer to tune buffer sizes for individual boards. When application code is modified to change message traffic, MIC can adjust to use only the necessary memory resources.

2.3.3 Command Preprocessor

The Command Preprocessor front-ends I/O originating from two sources: the driving Host computer (Host) and the operator's console (Console). At the lowest level, it drives the physical data links supporting these command streams. In the NavLab controller, RS232C serial channels are controlled. At the next level, it validates data integrity of Host-originated Virtual Vehicle Interface (VVI) command packets by checking format correctness, parameter count, and packet size. At the highest level it checks parameter values against established limits. The Command Preprocessor has the ability to reject commands exceeding the current operating limits, but the Pilot has final authority on command acceptance. Query commands issued by the Host are handled directly by the Command Preprocessor without Pilot involvement.

The Command Preprocessor communicates primarily with the Pilot module. The other modules are indirectly accessed through value lookups in the Scratch Pad. All commands involving action, such as motion commands or control commands to a device managed by the Bitbus Sensor/Control Network, are first sent through the Pilot to update its knowledge of the vehicle state affected by the controller.

The Command Preprocessor contains two separate subprocesses to service the Host and Console concurrently. The Host Interface is responsible for maintaining communications between the controller and the Host. The Console Interface interprets commands from the operator console keyboard. The Console is given priority over the Host so that it is possible for the operator to override Host commands. Commands are received as ASCII packets. The Host sends only numeric data; each command is given an opcode. The Console allows the operator to enter commands as simple mnemonics.

Communication errors are trapped by syntactic data validation. The Command Preprocessor takes two different actions based on the type of command it receives. For motion commands, the arguments are validated based on the allowable ranges of vehicle motions posted in shared memory by the Advisor. If all the arguments are acceptable, the command is passed on to the Pilot. An acknowledge message is then sent, signaling that the command was accepted and will be executed. If for any reason the command is found to be invalid, a disacknowledge message along with an explanation for rejection is sent to the command initiator.

The Command Preprocessor processes query commands (e.g., heading, position). The requests are satisfied by accessing the shared memory region where the information is updated constantly. This method makes it unnecessary to interrupt other processes. The data is formatted and shipped to the requestor.

The Command Preprocessor also maintains the display on the operator console onboard the NavLab. The screen is divided into three parts:

1. Display -- A window displays vehicle data. The operator can select between 5 different displays:
    - Sensor data shown in graphical form (vertical bars).
    - Sensor data shown in alphanumeric form.
    - Status of switches controlled by the controller shown in alphanumeric form.
    - Command packets between controller and the Host.
    - A help screen that explains how the operator can control the vehicle by using the Virtual Vehicle instruction set.
2. Command line -- Allows the operator to:
    - Enter Virtual Vehicle commands.
    - Enter software joystick commands.
    - Turn on/off switches controlled by the Bitbus network.
3. Information area -- A window is reserved for special messages that may be sent by any process in the controller.

2.3.4 Pilot

The Pilot's main function is controlling or initiating all physical action and motion control. The Pilot also plays the central role in inter-processor communications by acting as the hub in a star configuration. All commands altering the state of the vehicle are filtered through the Pilot, eliminating contention and state ambiguity problems potential to systems altered by multiple independent processes. For the generalized case, the Pilot module would occupy several processor boards and handle manipulation as well as locomotor control.

The Pilot is composed of a hierarchy of concurrent processes (tasks), each of which is dedicated to maintaining a specific subset of state variables and initiating all actions affecting those variables. At the lowest level, each axis of motion has an individual driver process associated with it that formats motor-controller specific command strings, performs I/O exchanges with the motor-controller board, and maintains the current values of all pertinent variables for that axis in local memory. The axis drivers at this level have no notion of the physical configuration of the overall system. Coordination of motions is handled by higher-level processes.

Action requests can be submitted to the Pilot by the Command Preprocessor at any time. On receipt of such a request, the Pilot returns an acknowledge/disacknowledge message to the Command Preprocessor indicating whether it can execute the command. If the received command can be executed, it is decoded and forwarded to the appropriate subprocess for handling. Depending on the type of action requested, this process may then

- direct motions (via the appropriate axis drivers)
- read or set parallel I/O lines (for example, to select a different transmission gear)
- update the values of some state variables.

Because individual processes each have a specific run-time priority, critical commands (e.g., "STOP") always obtain control of the CPU, even if a lower-priority command is still in progress. Also, because task scheduling is event-driven rather than time-shared, high-priority processes always run uninterrupted, i.e., in constant time.

A special set of tasks within the Pilot maintains and processes a queue of arcs specifying a path for the vehicle. These arcs are executed continuously and a position report is issued to the Command Preprocessor on completion of each arc. Velocity and acceleration parameters can be updated at any time during execution of an arc; in addition, one value for each of these variables may be queued to go into effect with the beginning of the *next* arc execution.

The Pilot has the final responsibility for command acceptance or rejection, command queue management, and implementing established equations to achieve requested arc trajectories. Implementation details of the vehicle are masked by the Pilot.

The NavLab incorporates braking as well as forward and reverse propulsion in a single, bi-directional hydrostatic drive. For the generalized vehicle case, the Pilot would coordinate brake/throttle control to achieve velocity and position objectives. At the servo level, motion is controlled by motion control boards commanded by the Pilot. Emergency stop conditions are signaled to the Pilot by a critical interrupt line controlled by a planned Health Preservation module with bump detection facilities. On assertion of this line, the Pilot is responsible for graceful shutdown, leaving the vehicle ready for recovery actions issued from the operator's console. Because only the Pilot controls the motion, it is always aware of the current motion state.

Finally, a few background processes perform such functions as maintaining the system clock and calculating position coordinates based on sensor measurements.

2.3.5 Sensor/Device Management System

Apart from the five main axes of motion, there are numerous sensors that must be monitored and devices that must be activated. The Sensor/Device Management System manages two classes of sensors. The first class is characterized by sensors and devices that need not be monitored/controlled frequently. For example, a sensor might be dedicated to monitoring hydraulic fluid temperature; while this information is important, it is not essential that it be updated more frequently than once in several seconds. Another class of sensors is that group of devices that must be monitored frequently. An example is a process that must analyze data from inertial devices and post these results in shared memory several times a second.

2.3.5.1 Bitbus System

The Bitbus System is a highly flexible and expandable data acquisition and control system. By taking advantage of the Bitbus distributed control architecture, the Distributed System supports analog status sensors and digital I/O channels using microcontrollers distributed on a serial network. Nodes on this network transfer data to the Bitbus Server module using the Bitbus message passing protocol. The Bitbus nodes are programmable to meet a wide range of sensor and control configurations. Data returned to the Bitbus server are conditioned and scaled at the Bitbus node, reducing computational requirements of the Bitbus server.

The primary responsibility of the Bitbus server is to acquire and move sensor data to shared memory locations recognized by other modules in the controller. When the Pilot sends an action command request, the server must format messages to control any devices supported by a Bitbus node. In support of these functions, the server must also handle node initialization, self-monitoring, and fault recovery for the Bitbus network. The chief advantage of using a Bitbus network is the modular expandability and flexibility that is inherent to the Bitbus architecture. Complex inter-processor message passing facilities are included in the architecture, relieving the programmer of much responsibility.

In simple systems with limited I/O points, the Distributed System could be replaced with a single board computer equipped with the appropriate I/O expansion modules. An effort should be made to keep I/O operations local to the processor to avoid consuming bus bandwidth. With either implementation scenario, the update rates of shared variables should be adjustable to control the bus access frequency of the Distributed System for tuning purposes.

The Bitbus network provides a distributed control structure to service the first class of sensors. A list of sensors and devices on this network can be found in Section 2.5.2.

The Bitbus network is based on a master (Bitbus server) and slave (Bitbus nodes) concept (Figure 2-4). Nodes provide the connection between the sensors/devices and the central Bitbus Server. Because each node operates independently, fast data acquisition can be achieved by distributing the work load among many nodes. Nodes can also be programmed to perform control tasks by reacting immediately to critical conditions as they arise.

The Bitbus Server, one process on the Sensor Management Module, initializes the network and monitors status. Because the nodes cannot initiate communications, the Server must continuously poll each node for output data. When the Server receives a message from a node, it posts the relevant information in shared memory for reference by other processes. When some high-level process needs to control a Bitbus node, a message is sent to the Server. This message is then broadcast on the network where it is

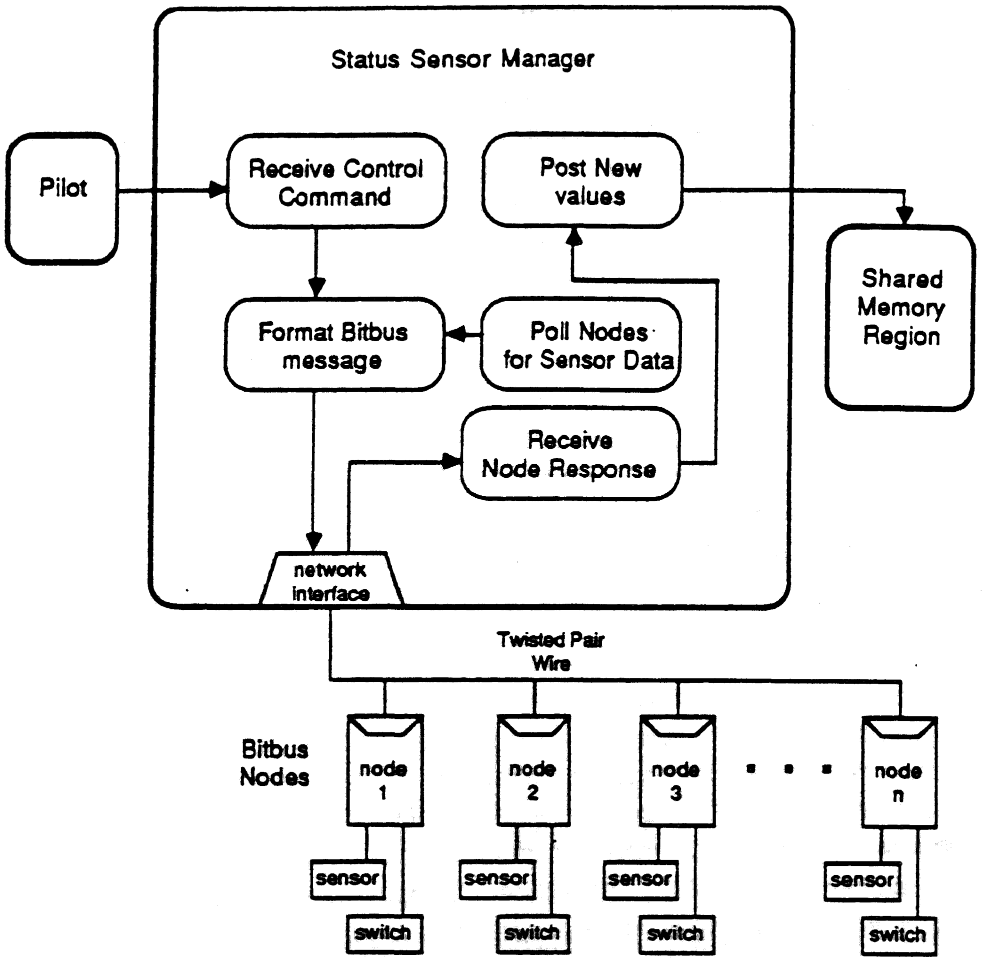


Figure 2-4: Bitbus Server

trapped and processed by the addressed node.

2.3.5.2 Fast Sensor Monitoring

The Sensor Management system also maintains processes to monitor those devices that must be serviced at a high frequency. At present, the only such device envisioned is an Inertial Navigation System anticipated to report position and orientation data about 10 times/sec. The incoming data is parsed and posted in shared memory. Other devices that need to be monitored constantly can be added to the controller simply by allocating a process to them. This method is preferable to the Bitbus method when data must be accessed frequently and must be made available to the entire system quickly.

2.4 Motion Control

Of the 5 axes of motion, only drive and steering can be controlled both manually and automatically. The other three motions of pan and tilt are only used in automatic operation. Figure 2-5 (a) shows the configuration during manual operation. All axes of motion on the NavLab are physically controlled by Galil DMC-200 series motor controllers. These controllers were chosen for:

- Multibus compatibility
- multiple modes of control (position, velocity, torque)

- coordinated motion of two or more axes (DMC-200 only)
- programmable acceleration and slew rates
- status, position, and error reporting.

A digital phase lead control law with adjustable gain, pole and zero provides a stable closed-loop system for a wide range of plant dynamics. The motor controllers communicate with the Pilot subsystem through Multibus I/O ports for data as well as handshake exchange.

Single axis Galil DMC-201 controllers are used for steering, drive, and laser-ranging pan motions, while a DMC-200 two-axis unit is used for the camera pan and tilt (Figure 2-5 [b]). Each controller is software-calibrated at power-up to match the dynamics of the controlled axis. Thus, motor controller boards can be interchanged simply by selection of appropriate I/O addresses via jumpers.

2.4.1 Dash Panel Control

The vehicle operates manually to simplify transport to and from test sites. Manual operation doesn't require any computing or generator power. The electronic components active during manual operation are powered by the NavLab's 12 V system.

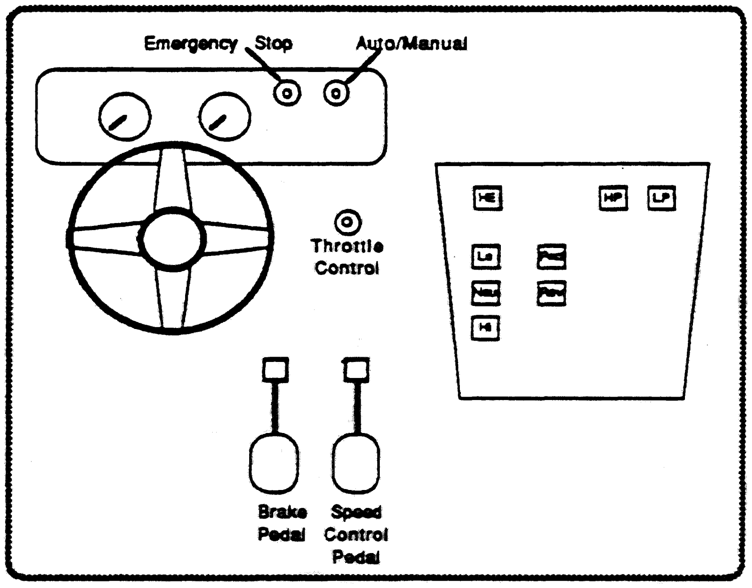


Figure 2-6: Dash Panel Layout

A human interface is incorporated for safe and easy use by drivers of standard automobiles. Figure 2-6 shows illuminated pushbutton controls mounted within reach of the driver.

- **High, Neutral, & Low**: allow the operator to choose gears. Because switching directly from one gear to another produces an unsafe lurching of the vehicle, a hardware logic function allows switching only by first selecting Neutral.
- **Forward, Reverse**: select the direction in which the vehicle moves.
- **Auto/Manual**: a pull-push switch that switches between manual and automatic control.
- **Emergency Stop**: disables autonomous locomotion and brings the vehicle to a rapid, controlled stop. Servo-lock of steering is disabled; steering is returned to manual control.
- **Brake Pedal**: as in commercial cruise control systems, a light touch of the brake pedal brings

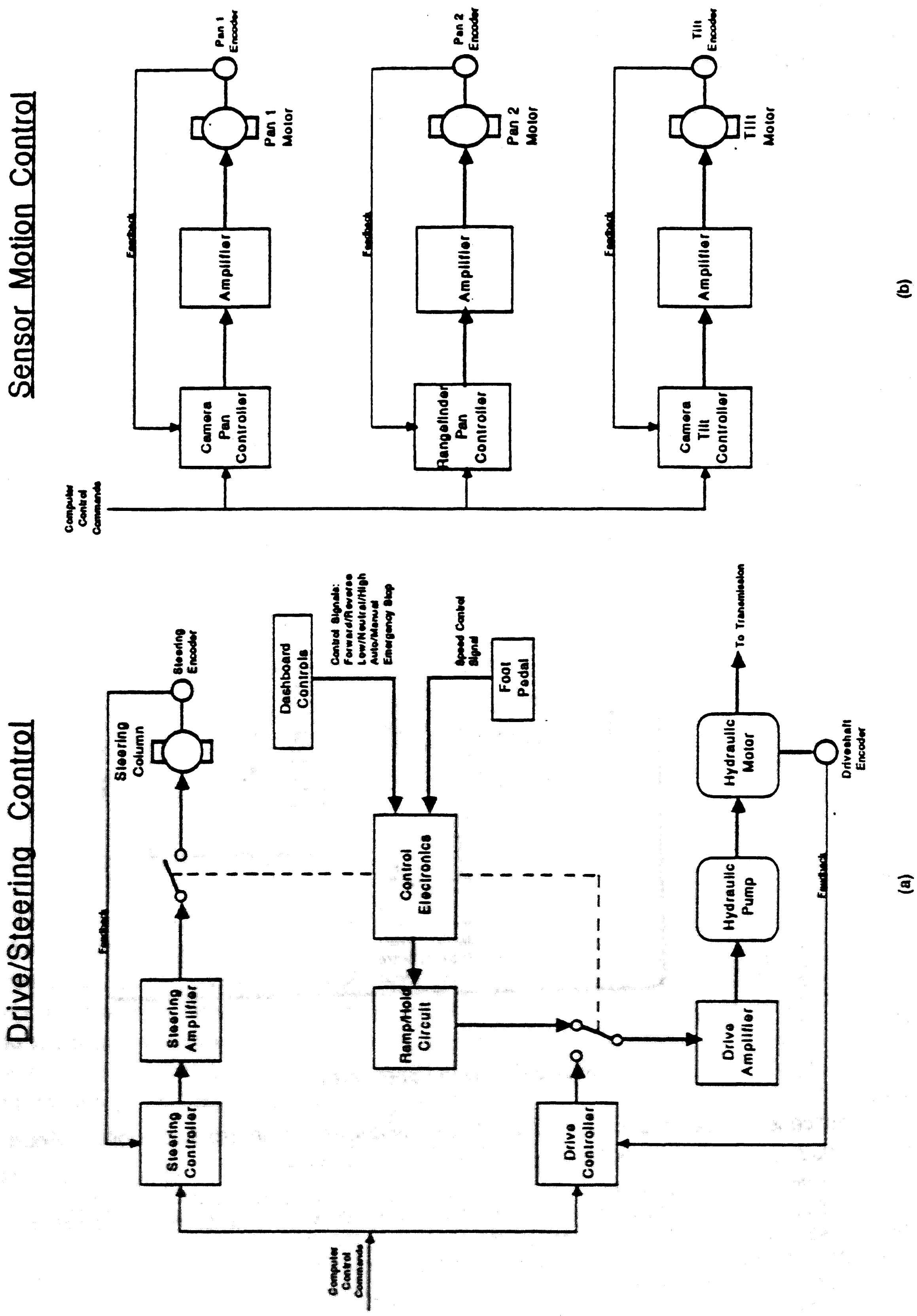


Figure 2-5: Motion Actuation

the vehicle back under manual control.

- **Speed Control Pedal**: activates a 20K ohm potentiometer to produce a voltage proportional to the angle of deflection of the pedal.
- **Throttle Control**: this dial sets the vehicle engine RPM as detailed in Section 4.2.1.1.
- **HE**: this switch turns on/off the heat exchanger fan for the hydraulic system.
- **HP, LP**: These lamps are lit when the dirty oil filters in the high and low pressure hydraulic systems indicate an alarm.

By default, when the vehicle is powered up, it is put into manual mode, neutral gear, and forward direction. It is necessary to provide the ability to override the autonomous mode in a fast but controlled manner if an emergency develops. To ensure reliable operation, manual override is a hardwired electronic circuit with sealed electromagnetic relays instead of sequential logic gates. This design proved to be immune to the noise and power fluctuations common to automotive electrical systems. Because this circuitry is essential to vehicle locomotion, it is powered by the vehicle 12 V system rather than the generator.

An electronic ramp/hold circuit in series with the foot pedal provides adjustable limits on acceleration and deceleration and ensures that abrupt movements of the foot pedal do not cause the vehicle to lurch. This feature was included both for safety and ease of driving. A second ramp/hold unit ensures a smooth deceleration in case of an emergency stop.

2.4.2 Steering Control

The steering control system consists of a computer-controlled DC servo motor linked to the steering column by a toothed belt. A single axis motor controller (DMC-201) uses feedback from an optical 1200-line incremental encoder mounted directly on the motor shaft to maintain tight position control over the steering wheel. A servo-amplifier converts the +/- 10V control signal from the motor controller to drive the DC motor with up to 11 amps of continuous current. At maximum speed, the steering mechanism can be moved between its two extreme positions in 2 seconds.

Feedback is obtained from an encoder on the motor shaft that is mated to the steering wheel, which is always turned a specified amount. Differences between intended and achieved radius occur due to linkage non-linearities and factors such as friction between the road and the wheels, grade of the road, vehicle speed, and speed with which the steering wheel is turned.

Limit switches on the steering linkage are hardwired inputs to the controller board and provide both a safety stop to protect the steering mechanism and a reference point for roughly calibrating the steering control system to a known position on power-up or system reset.

2.4.3 Drive

A single axis motor controller services the drive system. The voltage (-10V to 10V) produced by the motor controller is converted to a current signal (-100mA to 100mA) by an amplifier that directly operates a hydraulic servo valve to set the speed of the hydraulic motor. Acceleration of the vehicle is limited by a ramp/hold circuit, mentioned in Section 2.4.1, in effect providing a low pass filter to the input signal. An optical 300-line incremental encoder mounted on the hydraulic motor shaft provides feedback to the motor controller. Because the transmission is downstream of the hydraulic motor (i.e., between the motor and the driveshaft), the encoder pulses must be interpreted differently for high and low gears.

2.5 Sensors/Devices

At present, the controller features for handling sensors are not fully implemented. Two fronts of expansion are proposed for the near future. An Inertial Navigation System will be incorporated to provide continuous position and orientation information. A Bitbus network will be used to monitor and control devices distributed around the vehicle.

2.5.1 Inertial Navigation

An Inertial Navigation System (INS) to be deployed on the NavLab will receive distance data as input and will provide position and inclination data along the axes specified as output. The INS detects initial heading on its own and provides updates of position and heading.

The following information will be obtained from the INS:

1. True heading of the vehicle -- 0.5 degree resolution.
2. Rate of change of heading -- 0.5 degs/sec resolution.
3. X, Y, Z position in cm -- 10 cm resolution. This will allow movement on a 100 km^2 grid.
4. Roll and pitch inclination -- 0.5 degree resolution.

Performance criteria include:

- Dead Reckoning Capability: Speeds along the direction of travel of up to 60 km/hour; turning speeds (change of orientation) of up to 40 degrees/sec.
- Accuracy: Maximum long track error: 1% of distance traveled. Maximum cross track error: 0.1 degree/hour.
- Updates: Must be able to handle the accuracy requirements above with updates coming only once an hour or once in 5km.
- Necessity of Stopping: Must not need more than 5 minutes for the vehicle to be completely stationary on power-up or on recalibration.

At present a device that uses three mechanical gyroscopes and requires an odometer input is being considered. A second device being considered is a strap-down system that uses ring laser gyroscopes. This is much more accurate than the first and does not require odometer input.

2.5.2 Sensors/Devices on Bitbus Network

The following is a list of sensors and devices that are monitored and controlled by the Bitbus network. Scan cycle time indicates the period at which each of the sensors is monitored. Temperature units are degrees centigrade. Pressure units are pounds/sq. inch.

Sensor	**Scan Cycle**	**Minimum**	**Maximum**
Thermocouples			
Engine Oil	30 sec	0 deg	175 deg
Engine Coolant	30 sec	0 deg	150 deg
NavLab Cabin	30 sec	-10 deg	40 deg
NavLab External	30 sec	-10 deg	40 deg
Hydraulic Reservoir	30 sec	0 degs	100 degs
Pressure Transducers			
Engine Oil	10 sec	0 psi	60 psi
Hi-pressure System (input)	10 sec	0 psi	3000 psi
Hi-pressure System (output)	10 sec	0 psi	3000 psi

3. Vehicle Shell

NavLab's foundation is a 1985 General Motors Vandura cutaway chassis chosen as a commercial base to simplify development. As acquired, the vehicle consisted of a chassis, a drivetrain and a cab. A custom shell was constructed to house the onboard AC power generation, power distribution, control and computing equipment. Space for operators is provided, allowing research activity within the confines of the vehicle. The original configuration also included a 350 ci V-3, cruise control, an automatic transmission, dual rear wheels, power steering, power brakes and a 33 gal fuel tank.

3.1 Exterior Design

The shell was custom-built with particular attention paid to strength requirements, anticipating needs for extensibility. The roof and cab support air-conditioning, antennas, sensors, and working personnel. The floor of the shell supports about 2000 kg. The shell is dimensioned so that researchers can stand inside; five equipment racks are housed side by side along one side of the vehicle. Figure 3-1 shows a rear and side view of the vehicle.

Hie shell is made entirely of steel. Heavy gauge was used on the front and back walls while lighter gauges were used along the side walls and roof. A metallic blue paint protects the entire shell There are compartments for the generators and power-related equipment. Louvered metal doors provide outside access; there is no access to these compartments from inside the vehicle to keep fumes from entering the shel

A wiring port in the *floor* behind the driver's seat allows wiring from the underside to enter the vehicle. Another access vent in the shell above the passenger compartment enables wiring from cameras and range sensors to enter the vehicle.

3.2 Interior Design

Figure 3-2 shows a topdown view of the NavLab. The cab has two seats, one for a driver and one for a passenger. A console located between the two seats allows the operator to control and monitor the transmission. A research area behind the cab contains computing, sensing, and control equipment, as well as space for two researchers.

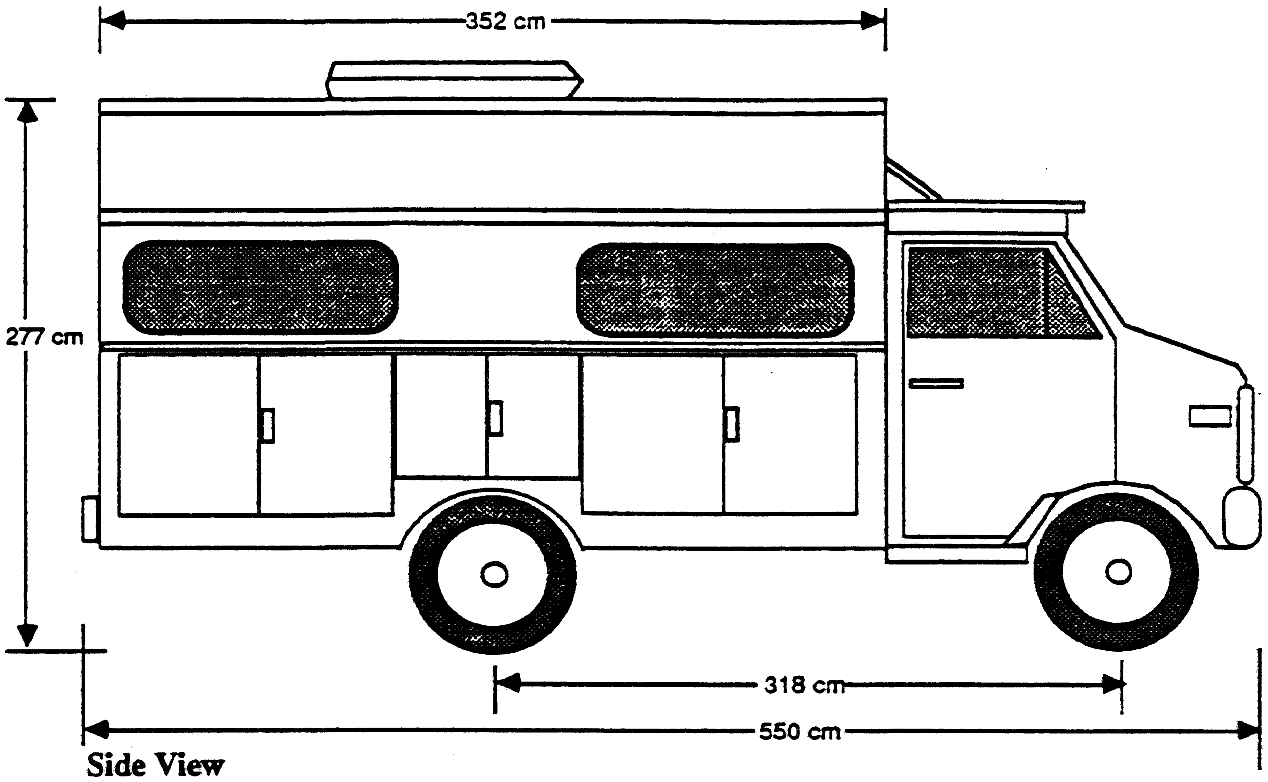
Five equipment racks are located on the left side. A desk area extending the length of the research area is located opposite the racks across the aisle. Three video monitors mounted above the desk area can swivel to a desired viewing position.

Along the rear edge of the desktop an outlet *strip* provides power for the various terminals ami test equipment. Soft-down inserts with elastic straps prevent computer equipment from sliding on the desktop when the vehicle is moving.

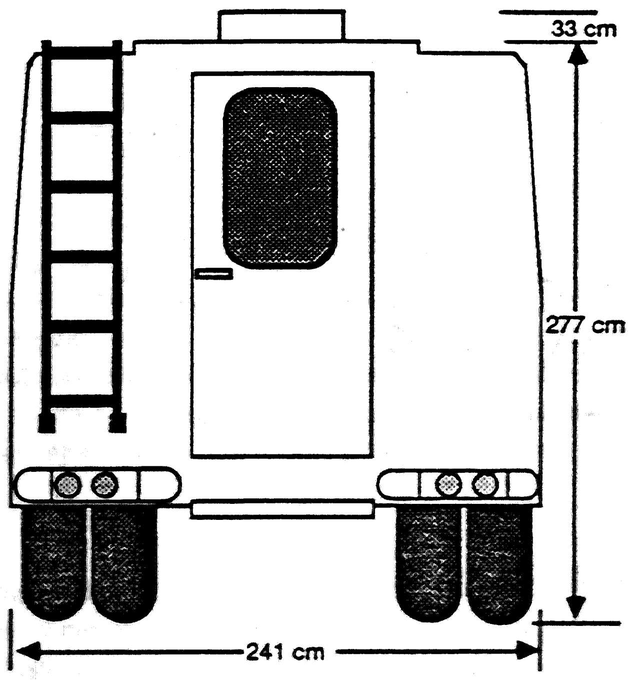
Cabling between devices passes through cable trays mounted close to the ceiling. The tray design securely holds video ami communication cables but allows for easy removal or addition. Track-mounted lights above the desk area provide independently aimed illiminartfon.

in addition to the two seats in the cab, a *swivel* seat$_9$ centered in the desk area, Is mounted on the wai of a generator compartment Extra removable seats can be mounted in the aisle for more researchers.

Voltages & Currents			
Battery Voltage	30 sec	0 V	15 V
RPM	2 sec	0 V	5 V
GAS Level	60 sec	0 V	5 V
Low Level Reservoir	60 sec	0 V	5 V
Swash Plate Angle	1 sec	-10 V	10 V
Switch Settings			
Transmission Gear State	1 sec		
Generators (2)	1 sec		
Heat Exchanger	1 sec		



Side View



Rear View

General Navlab Specs

Total Weight: 5449kg
Minimum Turning Radius: 750 cm
Center of Gravity: (112cm, 244cm)
(x,y,) with rear corner on the driver's side as the origin.

Figure 3-1: Side and Rear View of the Vehicle

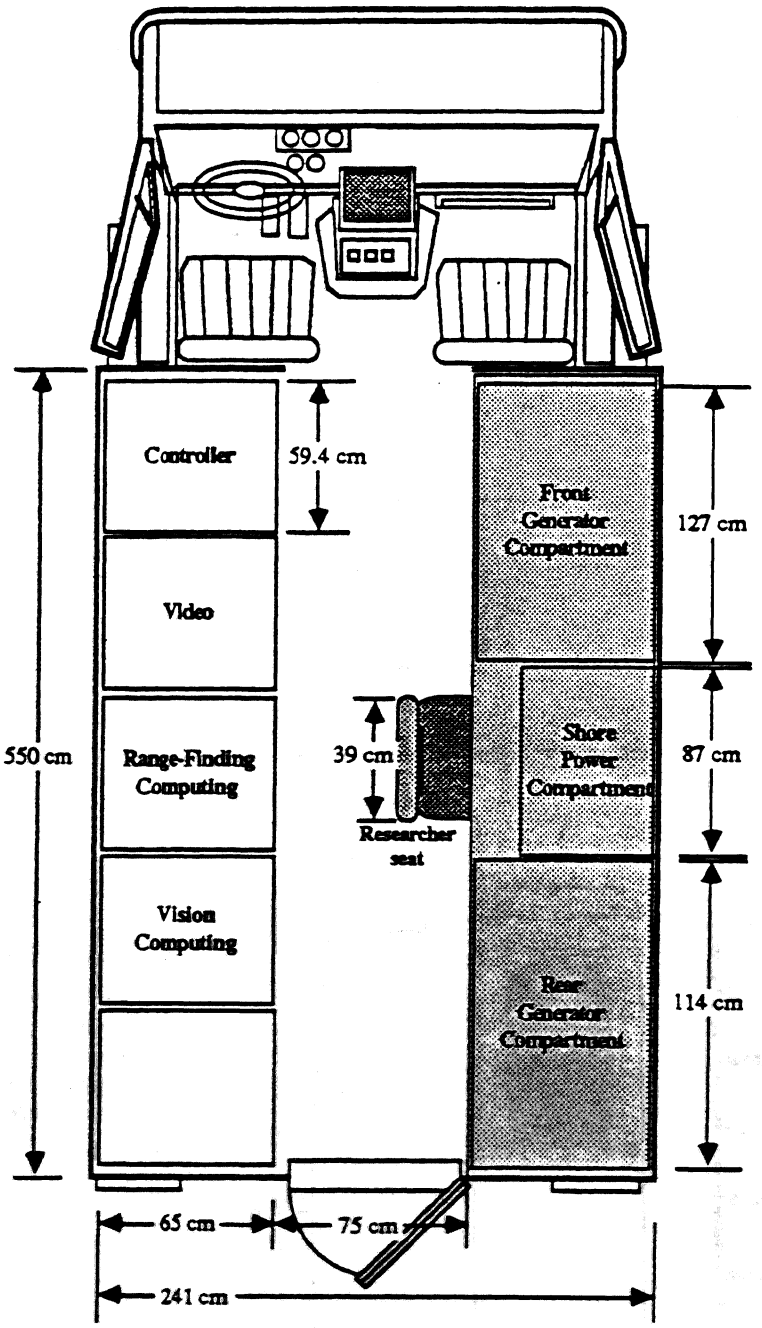


Figure 3-2: Interior Layout of Vehicle

3.2.1 Cooling

The heat generated by power conditioning, lighting, and electronics would damage some of the experimental computing. Thus in addition to the air conditioning provided on the van, a standard roof-mounted recreational vehicle air conditioner provides cooling.

4. Locomotion

Steering and drive motions coordinate to drive the NavLab through planned trajectories. Both axes of motion are controlled by analog signals issued by the controller while in automatic mode or through manual controls.

4.1 Steering

Figure 4-1 shows a front and side view of the NavLab steering mechanism.

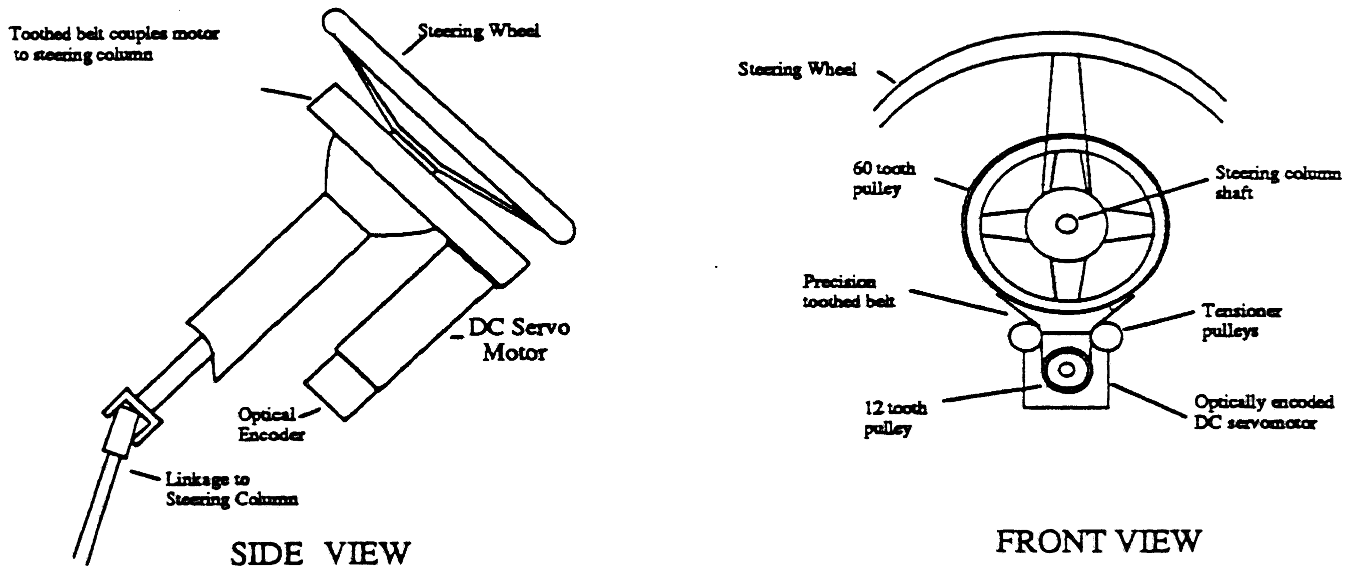


Figure 4-1: Steering Adaptation

The original linkage and steering column are driven by a DC servomotor mounted below the steering column. The motor is connected to the steering column shaft via a precision toothed belt; two toothed pulleys provide a gear reduction of 5 to 1. This configuration provides high enough torque to turn the steering shaft but low enough for the operator to overpower the steering motor in an emergency. A special hub ties all the steering elements together and a safety enclosure houses the moving parts. Limit switches at the extremes of steering travel prevent command error from damaging the system.

4.2 Drive

A hydraulic pump and motor combination comprises NavLab drive. This hydrostatic combination was selected because it provides precise control of position, speed and acceleration. Hydrostatic equipment also has a long history of smooth control and finely adjustable response.

Drive power comes from the main vehicle engine. Engine RPM is limited by a governor to prevent overdriving the attached hydraulic pump. Pump output is controlled by an analog signal.[1] The

[1]This signal originates from either a foot pedal that replaces the standard gas pedal or a drive controller, depending on whether the vehicle is in manual or automatic mode.

Fiber insulation between the shell and interior panels also provides protection from the heat. Insulation inside and outside the shell helps control interior climate. Underfloor insulation keeps heat from the hydrostats and exhaust from entering the interior. Flat sheets of fiberglass covered with thin gauge aluminum are inserted in floor areas between frame members. High temperature silicon-based insulation covered with heavy gauge aluminum foil covers exhaust pipes.

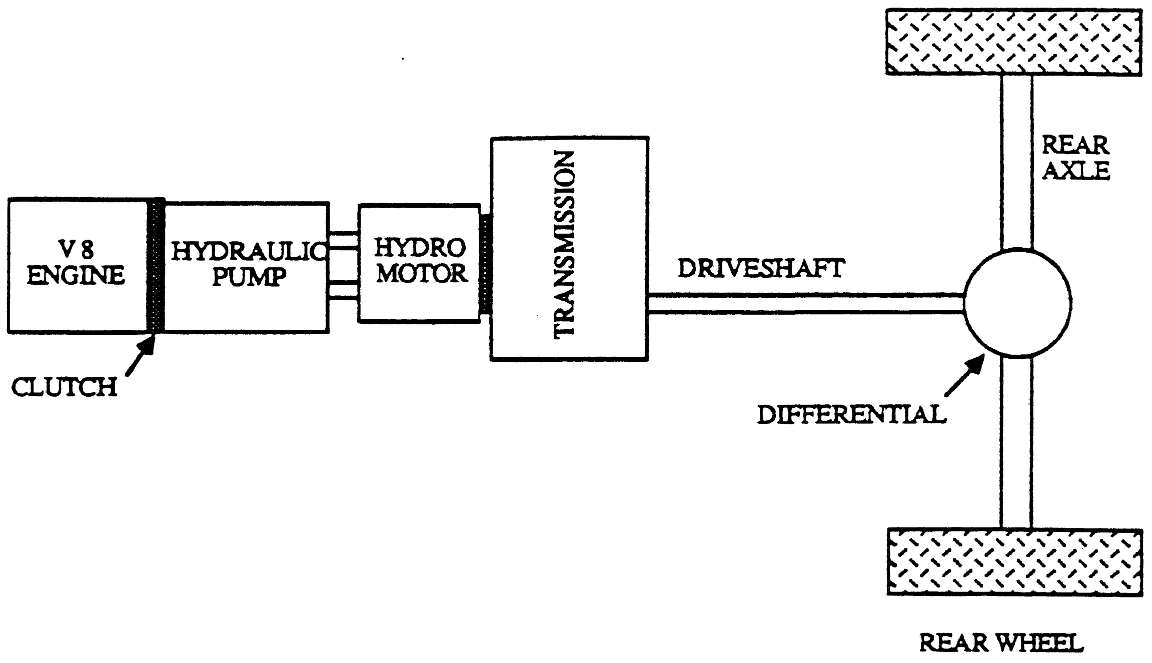


Figure 4-2: Schematic of Vehicle Drivetrain

displacement of the pump (proportional to the signal from the footpedal) determines the speed at which the hydraulic motor moves. The motor in turn powers a two-speed transmission which operates at either a 4:1 (low gear) or 1:1 (high gear) ratio, turning the driveshaft connected to the vehicle differential. Figure 4- shows a schematic of the drivetrain.

The configuration described above makes control of vehicle motion simpler than if the vehicle transmission and brakes had to be controlled to produce desired velocity. The standard braking system is intact but is only used in case of emergency because the analog signal to the hydraulic pump controls both accelerations and decelerations.

4.2.1 Vehicle Engine

The standard 350 ci V-8 engine is the main source of driving power. The following modifications were made:

- An electrically actuated clutch was installed to couple the vehicle engine to the hydraulic pump. The clutch is disengaged to isolate it from the engine when the engine is being started.
- The alternator was upgraded to a 120 amp dual output unit to satisfy the additional requirements of the two-battery, 12 volt system onboard.
- One stock emission control air pump was substituted by a hydraulic power takeoff unit. It is driven from the crankshaft end and shares a stock V-belt with the power steering pump.
- An engine oil cooler was installed to reduce oil deterioration caused by the constant high engine temperature.

4.2.1.1 Engine RPM Control

An engine RPM control keeps the vehicle engine running at a determined range of RPM irrespective of grade and speed. A magnetic pickup on the output shaft of the engine provides feedback to a specialized controller that maintains a constant RPM by moving an actuator linked to the engine carburetor. Figure 4-2 shows a schematic of the mechanism.

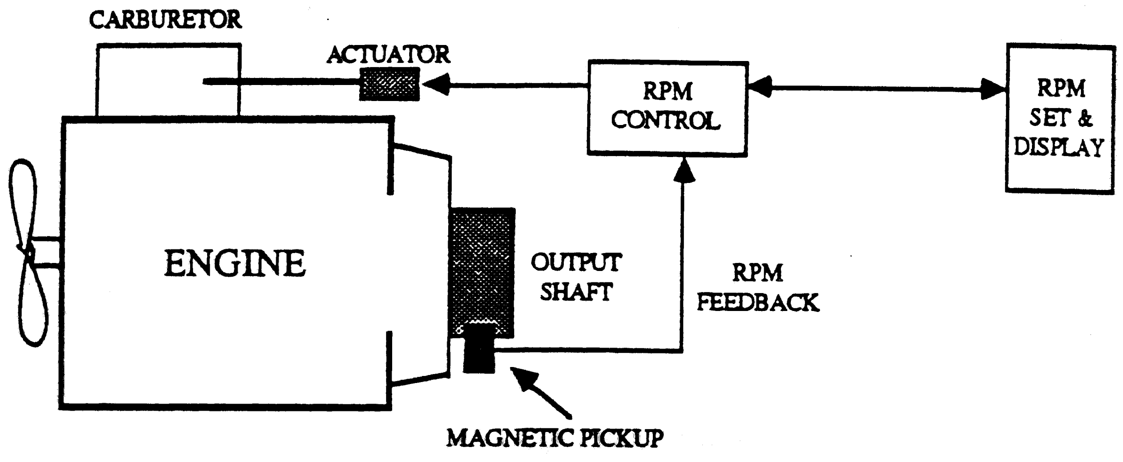


Figure 4-3: Mechanism for Engine RPM Control

4.2.2 Hydraulic Pump

The output of the engine drives a hydraulic pump through a flexible coupling. The pump is suspended from a frame crossmember with rubber shock mounts to allow movement with the engine.

The pump, a Sunstrand axial piston pump, is equipped with an electronic displacement control valve that alters the angle of an internal swashplate between 0 and 18 degrees, depending on an input signal that varies from -10 to 10 V. Negative voltages cause the pump to turn in the reverse direction. At 0 volts the pump has a holding torque to keep the vehicle stationary. At 10 V the displacement is maximum, corresponding to maximum driving speed.

Hydraulic fluid is supplied to the pump from the reservoir by an integral charge pump to replace the fluid pumped to the motor while an equal amount of surplus hot oil is drained from the pump case and passed through the main heat exchanger.

4.2.3 Hydraulic Motor

Hydraulic power from the pump is transmitted to a matching fixed displacement motor attached directly to the transmission. High-pressure flexible hose couples the motor and pump. Because the motor is a fixed displacement type, it always turns the same amount for every unit volume of fluid pumped in, resulting in an RPM of the motor that is directly proportional to the input signal of the pump.

A 10 micron filter cleans the return leg of the high pressure system. An additional crossmember supports the hydrostatic motor. The motor unit has an SAE standard shaft mounted to a mating flange on the driveshaft's forward universal joint yoke.

4.2.4 Transmission

The Funk transmission, an electrically shifted two-speed gear box installed between the hydraulic motor and the driveshaft, is bolted to a reinforced frame member. It is mechanically coupled to the motor on the input side and to the driveshaft on the output side. The transmission provides a ratio of 3.950:1 in low gear and 1.0441:1 in high gear. Low gear supports low-speed experimentation (0-20 km/h); high gear (0-40 km/h) transports the vehicle along public roads in manual mode. The gear is selected electronically by applying a voltage to one of two solenoids on the transmission; if neither solenoid is activated, the transmission is in neutral gear. It is necessary to shift into neutral when changing from one gear to another.

A flow-through lubrication and charge system was added to the transmission to circulate hydraulic fluid. The fluid is returned to the reservoir through a low-pressure filter.

4.2.5 Reservoir, Heat Exchanger, and Filters

A reservoir holds about 80 liters of hydraulic fluid. Because seals and bearing surfaces are sensitive to temperature and contamination of the hydraulic fluid, oil returned to the reservoir must be allowed enough time to de-aerate and cool. Heat is removed by passing oil from the pump case drain through a heat exchanger. Cooled oil is directed back to the reservoir. Dirt in the oil is filtered at two points: in the return leg of the high-pressure system and between the transmission and the reservoir.

A series of valves assist in the cooling and circulation of working fluid. A shuttle valve and a low pressure bleed-off valve act together to allow a small portion of the working fluid to circulate through the oil cooler and reservoir. A make-up pump replenishes the fluid that is removed via a bleed-off valve.

The reservoir is equipped with a thermistor and a level gauge to relay tank status to the vehicle controller.

4.2.6 Hydrostat Sensor and Control System

Figure 4-5 shows the sensing and control system associated with the hydraulic drive system. All the components are located on the underside of the vehicle so all lines enter the vehicle through a wiring port in the floor behind the driver's seat.

Control lines include:

1. Hydrostatic pump displacement: This line controls the swash plate angle in the pump regulating the displacement of hydraulic fluid to the motor.
2. Gear selection: This line controls the gear (high or low) of the transmission.
3. Heat exchanger fan control: This line controls the on/off state of a fan that cools the hydraulic fluid.

Sensors include:

1. Dirty filter sensors: one dirty filter sensor is installed in each of the high- and low-pressure legs of the hydraulic system. These sensors trigger an alarm when they become clogged.
2. Pressure transducers: These read system pressure at input and output of the hydraulic motor.

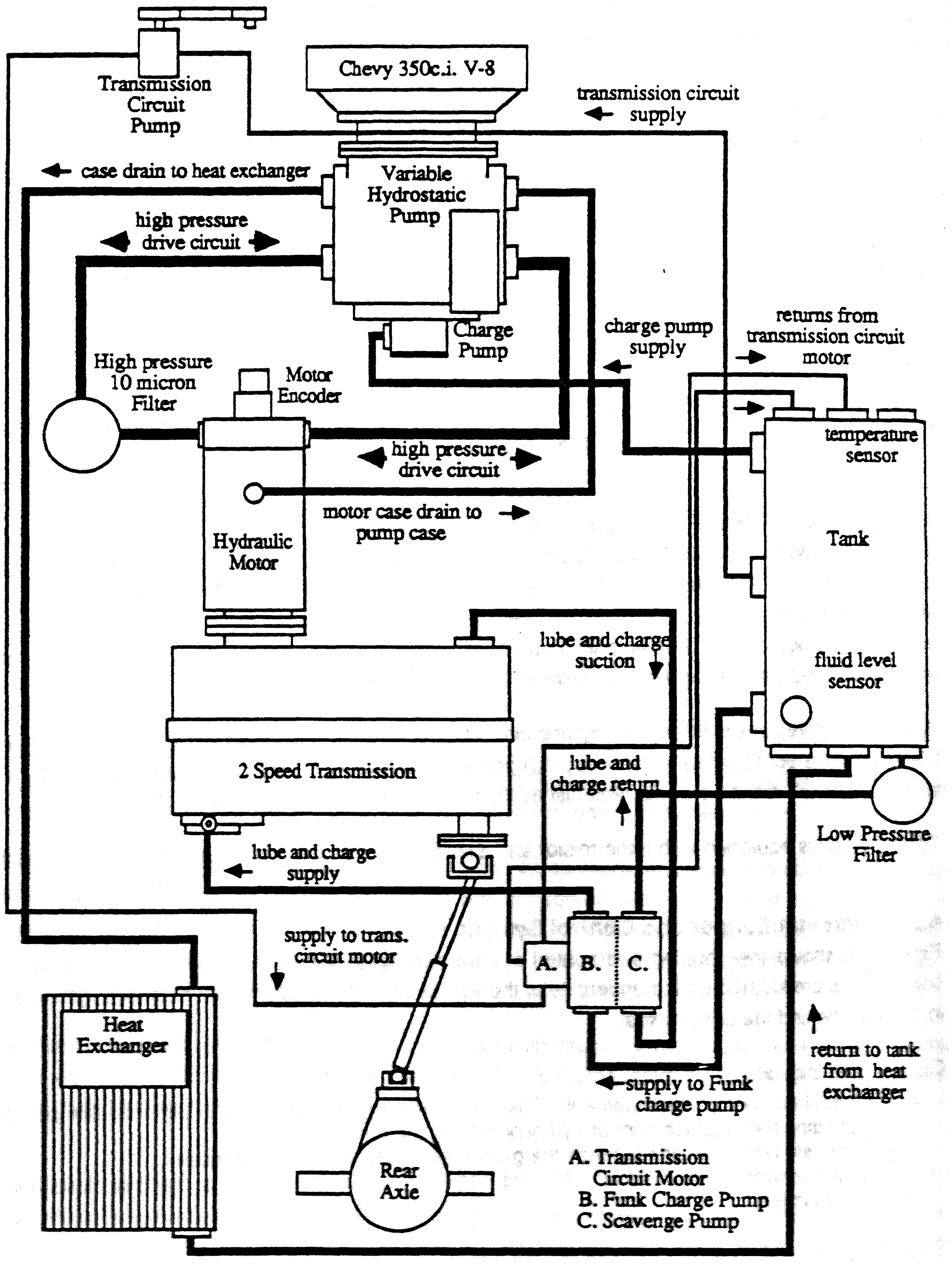

Transmission Circuit Pump
Chevy 350c.i. V-8
transmission circuit supply
case drain to heat exchanger
Variable Hydrostatic Pump
high pressure drive circuit
Charge Pump
charge pump supply
returns from transmission circuit motor
High pressure 10 micron Filter
Motor Encoder
temperature sensor
high pressure drive circuit
motor case drain to pump case
Hydraulic Motor
Tank
lube and charge suction
fluid level sensor
2 Speed Transmission
lube and charge return
Low Pressure Filter
lube and charge supply
supply to trans. circuit motor
A.
B.
C.
Heat Exchanger
supply to Funk charge pump
return to tank from heat exchanger
Rear Axle
A. Transmission Circuit Motor
B. Funk Charge Pump
C. Scavenge Pump

Figure 4-4: Hydro Drive System

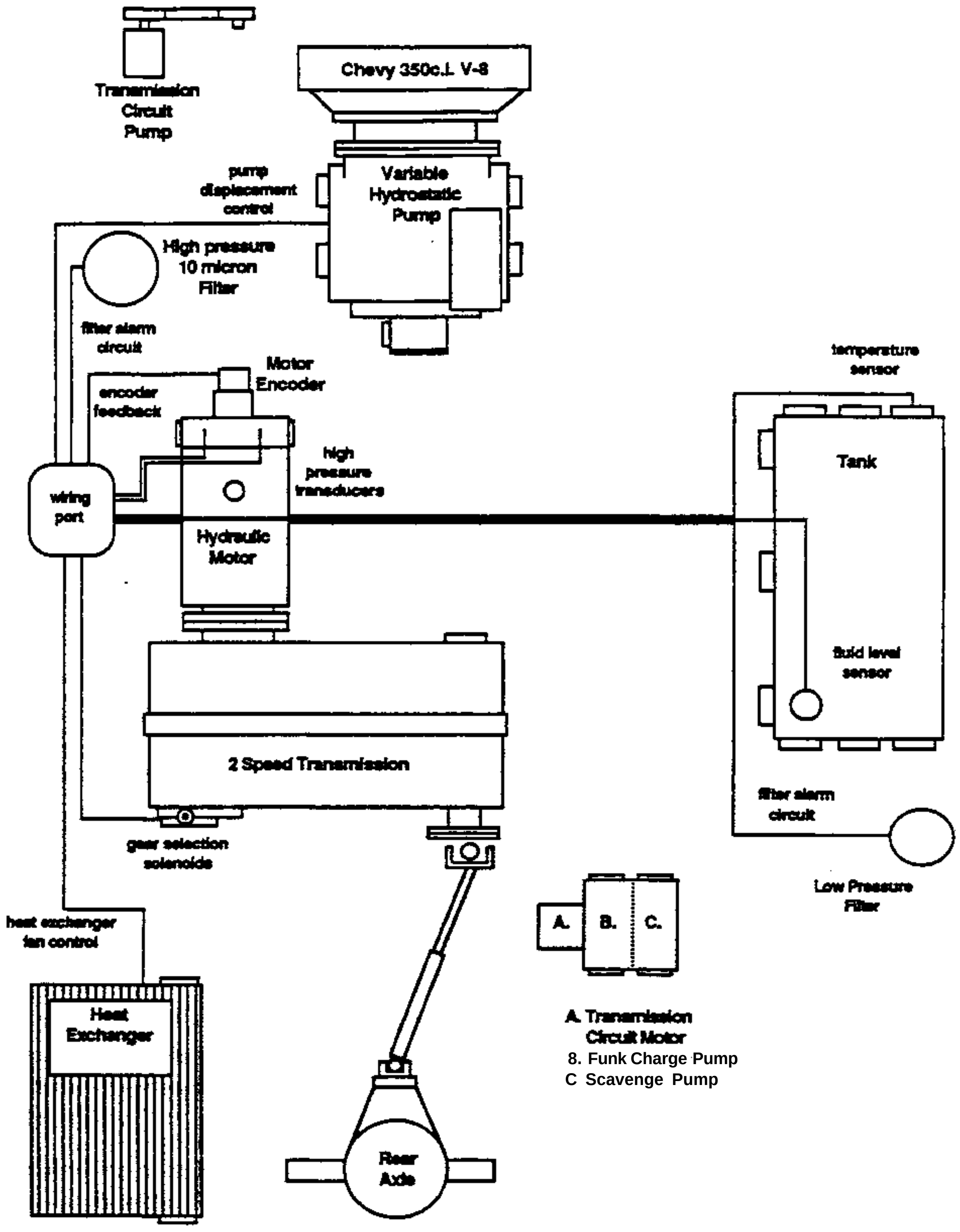


Figure 4-5: Hydrostat Sensor and Control Unes

3. Reservoir sensors: These measure fluid level and temperature of hydraulic fluid in the reservoir.
4. Motor encoder feedback: An optical encoder mounted on the shaft of the hydraulic motor provides feedback to the drive controller.
5. Steering limit switches: Limit switches are installed on the steering mechanism to signal an alarm if the wheels are cranked beyond acceptable limits.

5. Electrical System

All the electrical power needed by the NavLab is available onboard the vehicle. Electrical power can, however, be brought in from a shore power plug while the vehicle is in a fixed location. Power is distributed such that the generators are not needed to drive the NavLab manually.

5.1 AC Power

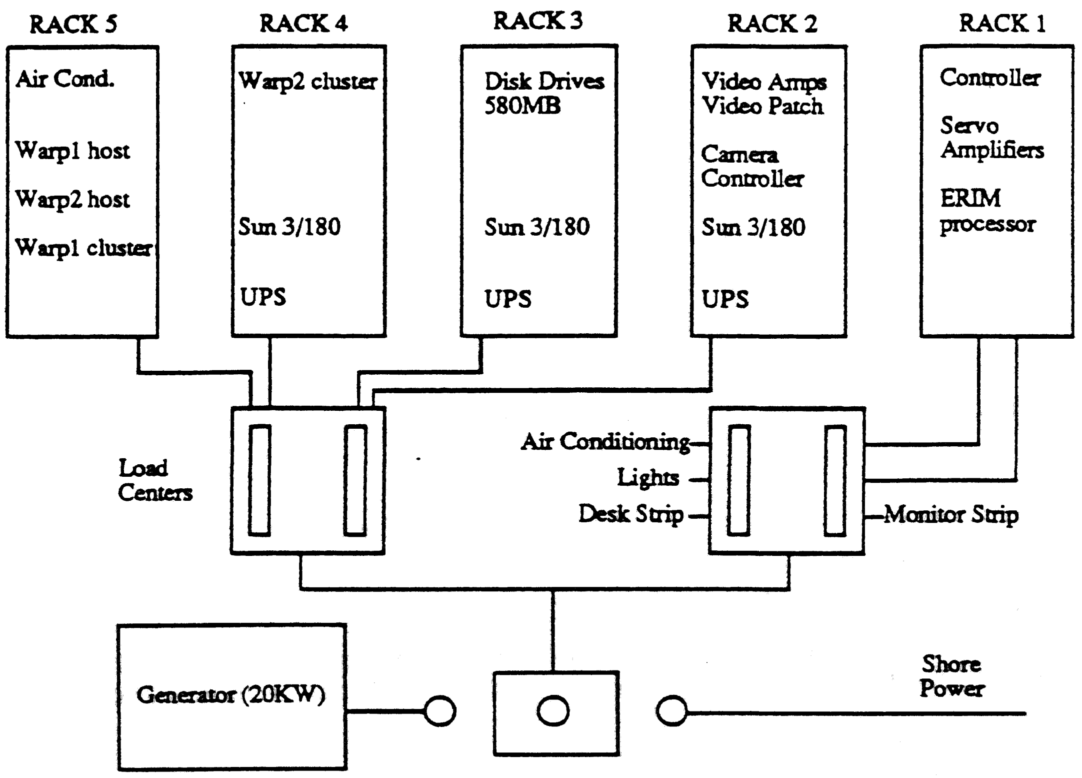


Figure 5-1: Wiring Schematic for AC Power

Figure 5-1 shows a schematic of the AC power system onboard the NavLab.

5.1.1 Generators

The generator supplies 100 VAC power to the variety of devices on the NavLab. The generator resides in a compartment accessible only from the outside of the vehicle, insuring the separation of noxious fumes from the interior.

Two compartments house an engine that is hydraulically coupled to a hydraulic generator unit in the forward compartment. This arrangement allows a single source of power up to 20 KW. Fuel to supply the engine comes from the vehicle fuel lines and the electrical power to start the generator is supplied by the vehicle 12 VDC system. The unit can be stopped and started by a panel switch.

5.1.2 Shore Power

The NavLab can plug into power from a building when stationary, alleviating constant generator operation. An extension cord from a nearby power outlet (220 VAC 50A) mates to a 220 VAC single-phase plug mounted in the outside center compartment.

5.1.3 Power Conditioning

Because variations in load and temperature affect generator power output, the power from the generators must be be conditioned to protect sensitive machinery from spikes and brown-outs. This is done by passing power through Uninterruptable Power Supplies (UPS). These devices not only condition the power from the generators but also provide full-load backup for up to 15 minutes, even if the generators or shore power are shut down, allowing a graceful system shutdown if power fails. Three UPS devices provide a total of 6KW of conditioned power, which will more than suffice for a complete configuration of computing equipment. The lights, air conditioner, video monitors, and servo-amplifiers do not receive conditioned power because they are much less prone to fluctuations in generator output.

5.2 DC Power

Because many of the devices onboard use DC power, the standard vehicle 12 V system was extended by adding an extra battery and replacing the alternator with a dual output 120 amp unit that charges both batteries.

Figure 5-2 shows a wiring schematic for the DC power system. The original battery powers:

1. Vehicle ignition -- starting power for the engine;
2. Dash panel -- all switches on the dash panel;
3. Interior lights -- overhead lights in the research area;
4. Control electronics -- the input voltage to two power supplies. One converts the 12 V into -12 V and the other converts 12 V to 5 V. Each power supply has a limit of 3 amps.

The second battery provides:

1. starting power for the generators;
2. power for generator compartment lights;
3. power to run the hydraulic fluid cooler fan.

An additional 28V power supply is mounted in the equipment racks. This takes an input from the 110 AC system and produces up to 30 amps of current, most of which is used by an ERIM laser scanner. The inertial navigation device will operate on the same power supply.

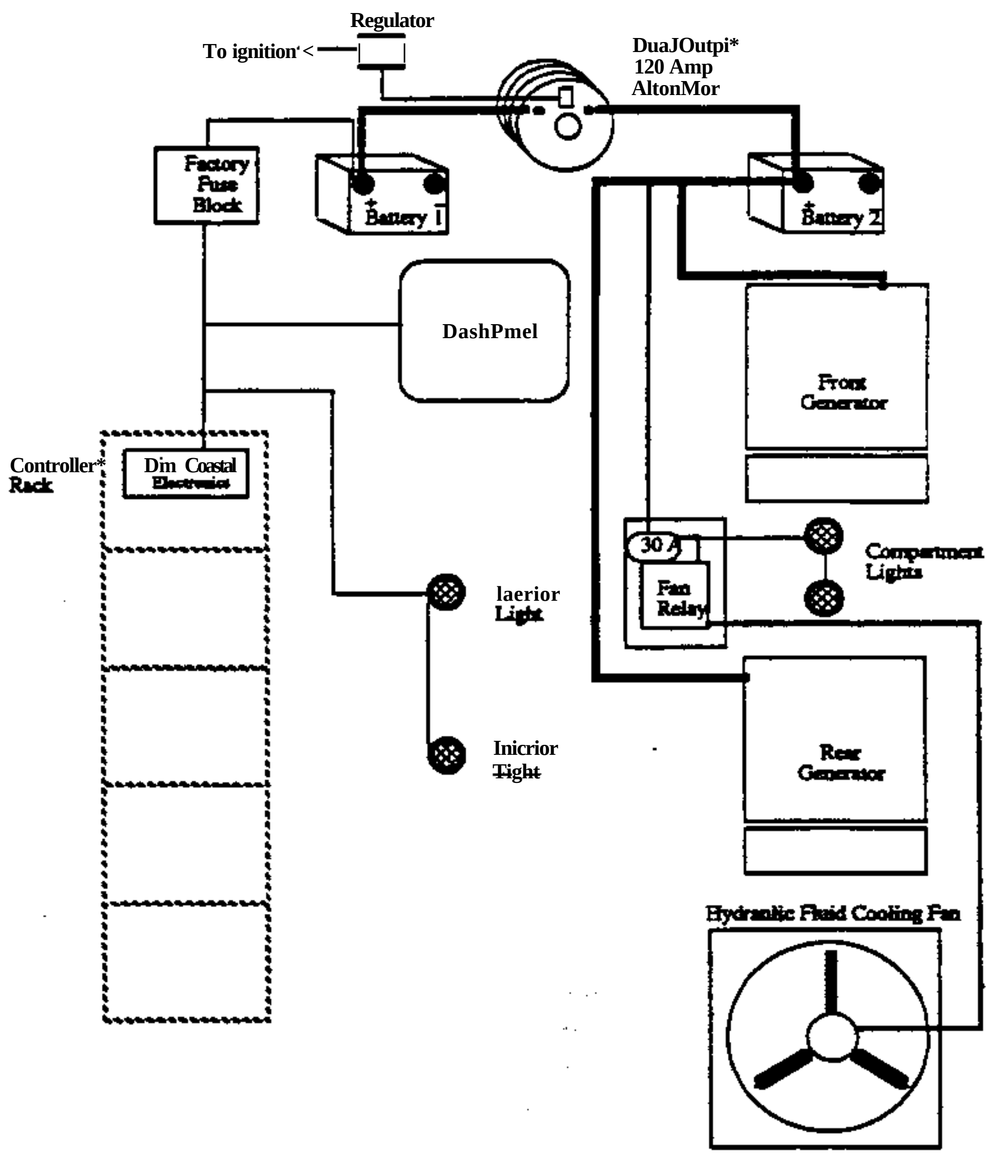


Figure 5-2: Wiring Schematic for DC Power

6. Telemetry

Telemetry to the vehicle was thought to be useful where environmental sensitivity, location, or size of computing equipment precludes installation onboard the NavLab. This feature has not yet found use in practice. NavLab telemetry provides control and monitoring from a remote site, allowing stationary computers to be used in navigation experiments.

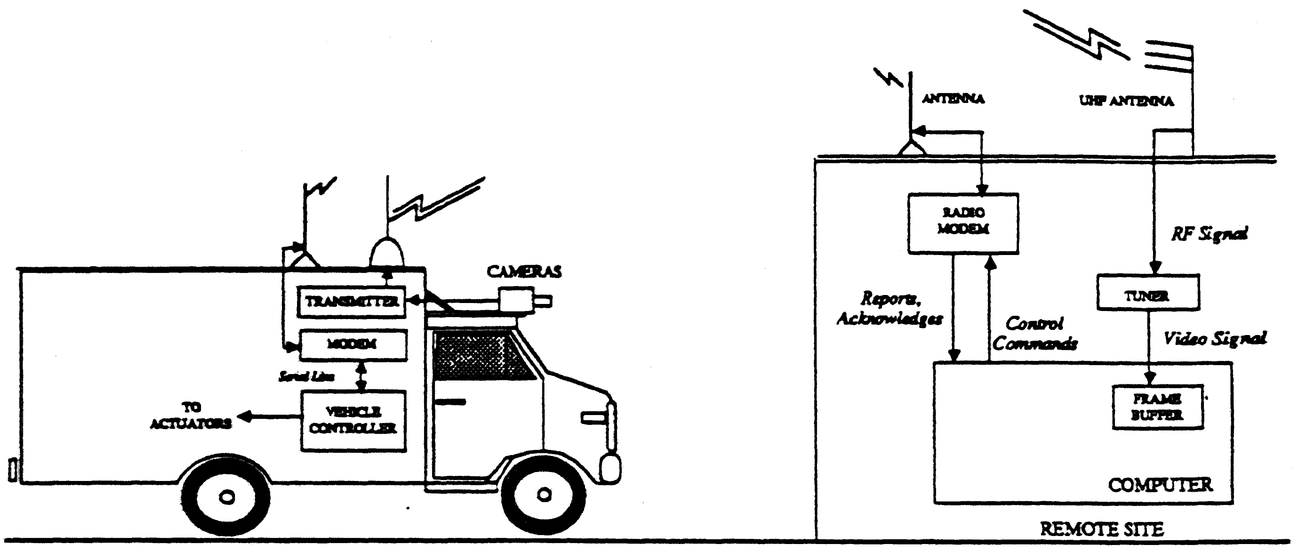


Figure 6-1: Telemetry Configuration of NavLab

The scenario in Figure 6-1 shows the closed loop set-up of a vehicle experiment where computing might be distributed offboard. The camera outputs a video signal that is broadcast over a UHF frequency and picked up by an antenna and receiver located on the Carnegie Mellon campus. The receiver provides the video signal to a frame buffer within the computer that processes the image. The signal is digitized and then analyzed. Commands to the vehicle are sent over a serial line to a wireless modem. A radio modem on the vehicle picks up this signal and feeds it to the controller.

6.1 High Bandwidth Transmission

An experimental radio license obtained from the FCC covers several broadcast frequencies. The license covers 2 UHF television channels, a full duplex radio link, and a 2 MHz microwave link.

The video signal is transmitted on the video transmitter while range data are transmitted over an aural sideband of one UHF channel. Because transmission rates can be as high as 56 K baud, the other aural sidebands not currently in use could serve several other data transmission needs.

6.2 Low Bandwidth Transmission

Two sets of 1200 baud radio modems are used for simple, low-bandwidth digital communication. These devices provide a transparent RS-232 connection between computers and facilitate sending commands to and from an offboard machine and the NavLab controller.

6.3 Cellular Phone

Separation of vehicle from stationary base facilities involves not only communication over distance for the machines but also for humans. For this reason, a cellular phone using existing mobile communications networks within the city has been installed.

7. Perceptive Sensing and Computing

A variety of sensors can be mounted on the NavLab depending on the type of research being conducted. Vision, laser ranging, and sonar ranging are the most popular sensing modes. More details of sensors and computing can be found in articles listed in Appendix V.

7.1 Video

Cameras provide a standard RS-170 video image to the frame buffer. Vision processing transforms this image into a scene description to support navigation.

Typically, a single camera mounted in the front of the vehicle provides a wide-angle view of the scene. Some vision algorithms, however, call for a stereo pair of cameras. Broadcast-quality cameras that provide red-green-blue color signals are used. Remote control units allow control over camera functions like gain, color balance, and iris size. Presently, camera focus and zoom must be controlled manually.

7.2 Laser Ranging

Laser ranging is useful in areas where vision algorithms fail -- in detecting depth discontinuities in scenes where the edges are not obvious and in those scenes that have uneven lighting because of shadows. Whereas the camera is a passive instrument, the laser rangefinder is an active device that emits a beam in a raster fashion and captures the reflection to provide two types of information -- distance and reflectance. The data are then analyzed to provide a scene description. Laser ranging provides a direct 3-D description of the scene while vision requires more expensive computation to extract this information. Range readings are particularly useful because they are not affected by ambient light.

The current laser ranging device, manufactured by ERIM, provides a 256x64x8 bit depth map. The scanner output is processed by a Motorola M68000 processor and sent to a Sun computer dedicated to ranging.

7.3 Pan and Tilt Mechanism

Vision and ranging sensors can be mounted in various configurations. Most configurations call for two independent pan motions -- one for the laser scanner and another for the cameras. Tilt is needed for both the laser scanner and cameras.

Pan and tilt design reflects a need to accurately position sensors over a large viewing range. Less than 4 seconds is required to view 180 degrees.

Cameras are mounted on rigid 5 cm diameter aluminum poles 2 meters long mounted horizontally through a worm drive gearset with a hollow bore. The gearset provides a 50:1 ratio and is driven by a DC brushed servo motor. An 800 line encoder provides feedback for the tilt motion.

7.4 Computing Configuration for Sensing

Figure 7-2 shows configuration of computing for simple, perceptive sensing. Much more complex setups are common. Each sensor commonly requires its own workstation or specialized processor. Another computer runs the blackboard system that integrates perception, modeling, and planning.

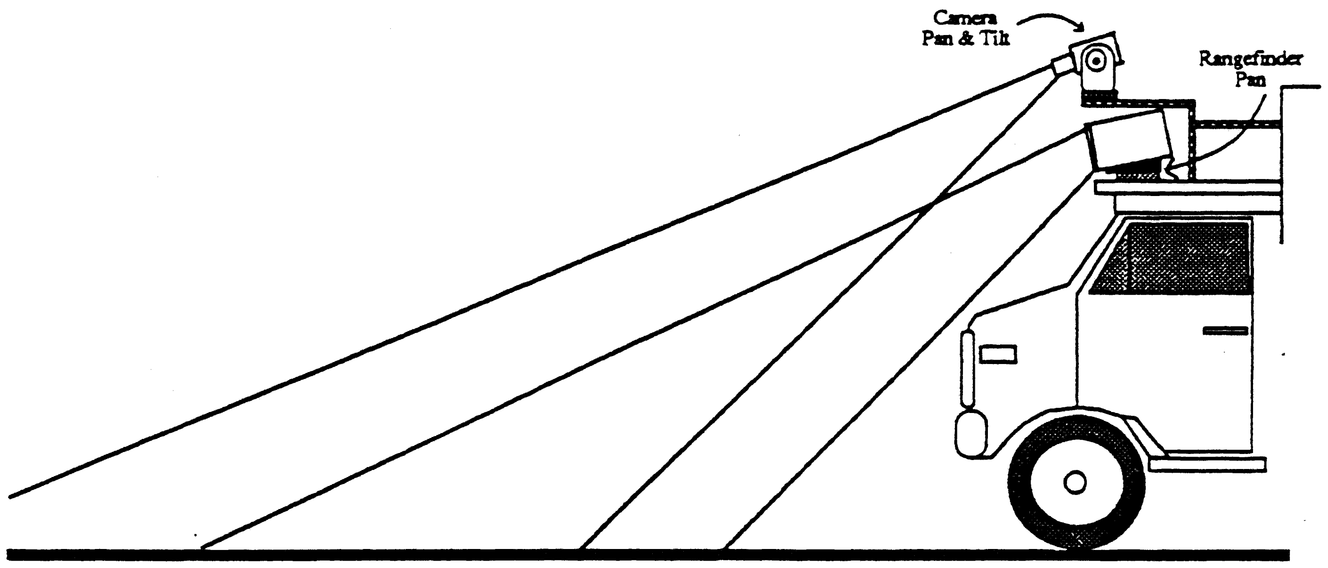


Figure 7-1: Pan and Tilt Mechanism

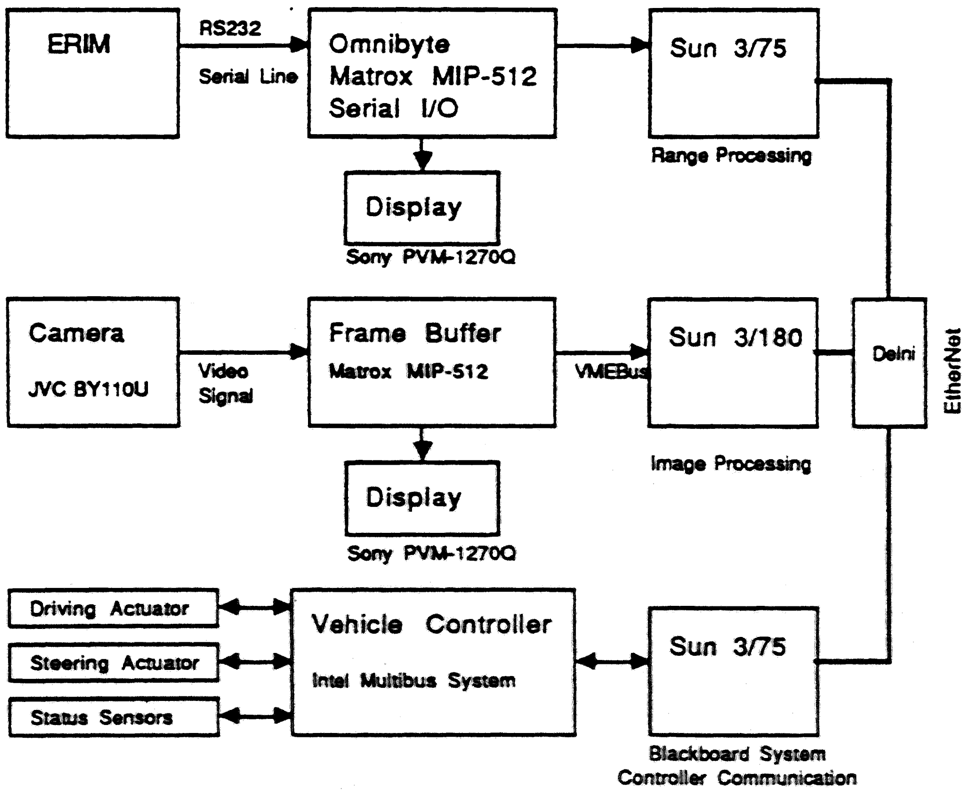


Figure 7-2: Typical Architecture

I. Modifications to Vehicle

The chassis was originally rated for 10,000 lbs. gross vehicle weight. The final vehicle weight was established as 12,000 lbs., thus necessitating a more robust suspension. In order to achieve the necessary load rating the front coil springs were upgraded and two extra leaves were added to the rear springs, increasing the gross vehicle weight by 2400 lbs. Heavy duty gas/oil shock absorbers were installed to minimize a slight tendency to pitch due to the extra weight. In addition, the original equipment tires were exchanged for Goodyear radials with a higher load and all-weather rating.

Modifications to the frame were minimized to preserve strength and stiffness. However, in order to mount some of the larger hydrostatic components alterations had to be made. The main forward crossbeam, transmission rear support beam and surrounding floor were completely removed. The crossbeam was replaced by a box section which bridges the hydrostatic pump. A channel section was added as support for the pump and also provides additional frame strength.

IV. Implementation of the Virtual Vehicle Instruction Set

Protocol

A simple high-level handshaking protocol has been designed for RS-232 communication between the Virtual Vehicle and any host computer issuing commands to the vehicle. It provides a means of exchanging commands and status information with a reasonable level of reliability and optional error recovery. The motivation for this is that errors in communication should be detected and acted upon without interfering with normal operations of host or Virtual Vehicle. By adhering more or less strictly to the protocol, the relative importance and subsequent computational overhead of error-free communication can be chosen at will and may be varied dynamically.

Messages between Virtual Vehicle and host are of the following form:

```
<length><packet ID><opcode>[<argument 1>/../<argument n>/]<CR>
```

The individual fields of a packet are defined as follows:

`<length>`	2 characters wide. Contains the total number of ASCII characters in the packet, including the length field itself. Length is represented in decimal, so messages are limited to 99 characters.
`<packet ID>`	3 characters wide. Unique identifier to be used as reference to the packet in subsequent protocol transactions. Can be any combination of printable ASCII characters (20H - 7FH), although numeric values (30H through 39H) will be used most frequently. See description below for usage and purpose of this field.
`<opcode>`	2 characters wide. Represents, in decimal, the numeric index of the command to be acted upon by the Virtual Vehicle. This implies a range of 0 through 99 for possible opcodes.
`<arguments>`	Zero or more numeric arguments, of variable width. Arguments are terminated by a slash ("/"). Leading zeroes are allowed. All arguments must be integer values.
`<CR>`	Carriage Return (0DH) character indicates end-of-packet. Must follow immediately after the last argument-terminating slash, or after the opcode in the case of zero arguments. The `<CR>` is NOT considered part of the packet, so it is not included in the `<length>` field.

The following rules define the handshake between two devices. They should be followed closely to achieve maximum communication reliability. However, as is indicated in the appropriate paragraphs, error checking is done at the discretion of the receiving device.

- For every packet to be sent, the sending device generates a unique 3-character packet ID code. This can be done, for example, by incrementally numbering packets or by encoding the current system time. Random generation of codes is discouraged, since this theoretically allows duplicate packet ID's. Using the full range of 96 symbols in each of the 3 character positions yields a range of 884,736 unique ID codes, which is in excess of the anticipated number of messages exchanged during a typical mission of the Virtual Vehicle.
- Packets are prefixed with the length of the packet and terminated by the end-of-packet character <CR>. The receiving device should check the actual length of the received packet against the `<length>` field to ensure integrity of each packet.
- Each packet received may be further validated by ensuring that
  - the opcode is valid,
  - the number of arguments is correct for the given opcode, and
  - the arguments are within allowable limits. These limits may change dynamically as a

Rack 1:			
	UPS		54.4
	2 Amplifiers, 1 Transformer, 3 Power Supplies		52.2
	Intel Chassis		24.9
	Patch Panel & Control Circuitry		9.1
	Tables		11.3
Total		42,287	152.0
Rack 2:			
	Video		22.7
	ERIM Power Supply		24.0
Total		42,226	46.7
Rack 3:			
	Sun		68.0
	UPS		54.4
Total		42,165	122.5
Rack 4:			
	Sun		68.0
Total		42,104	68.0
Rack 5:			
	Empty		
Total		42, 43	0.0
Pan & Tilt			
	2 1135 Motors		8.9
	2 Turntables		26.3
	1 Tilt Gearing		22.7
	1 Aluminum Rod		4.5
	2 JVC's		5.0
	ERIM		34.0
	PMI motor		4.5
	56 C Coupling		4.5
	Bracketing		
	lower mount		18.1
	upper mount		45.4
Total		121,447	174.0
Shell		121,171	1153
Van Body			
	front axle	101,427	1120
	rear axle	101,109	649

Overall:
Center of Gravity: 112,244 Weight Total: 5449kg

function of, for example, vehicle speed or road conditions.

- If none of the above error conditions are detected, the receiving device returns an ACK message to the sender, indicating that the message was received correctly and the appropriate action, if any, is being performed. The ACK message uses the *same* packet ID as that of the message being acknowledged.

- In case of an error, the receiving device must return a NAK message to the sender to indicate that the packet was rejected and no command is being executed. As its only argument, the NAK message contains an error code indicating the reason for rejecting the packet. A NAK message also has the same packet ID code as the message in question.

- Upon receiving a NAK as reply to a message, a device has the option of retransmitting the offending message (with new packet ID), logging the error, ignoring it, or taking any other action that might be appropriate. By the same token, while expecting a NAK or ACK in response to a transmitted message, a device may choose to time out, wait forever, or take other appropriate action. These conventions provide for very flexible operation that allows critical system operations to continue even in case of protocol errors. In dealing with these situations, the Virtual Vehicle will adhere to the following conventions:
    - NAK or ACK messages are *always* generated and sent in response to data messages received by the VVI. At this point, handshake for the current message is considered complete; i.e., no further action is *expected.*
    - Unexpected NAK or ACK messages (i.e., those referring to an unknown or previously acknowledged packet ID) are ignored. However, the error *is* logged and/or announced at the VV system console.
    - If a NAK is received as response to a message originated by the VV, the message may or may not be retransmitted repeatedly (with a new packet ID), depending on the type of message and reason for rejection.
    - If neither NAK nor ACK is received by the VV within a certain timeout period (configurable parameter, typically on the order of seconds), the error condition is logged and/or announced at the VV system console. After this, the VV still expects a response to the packet in question, but no further action is taken and subsequent messages are treated as if no error had occurred.

Commands issued by Host/Console

Mnemonic	Opcode	Meaning	Arguments
NAK	00	Negative Acknowledge	cc/ cc= 00 : Packet length error 01 : Num. of Args error 02 : Not ready for cmd 03 : Illegal Opcode 1X : X'th argument below current minimum limit (1 < X < 9) 2X = X'th argument above current maximum limit (1 < X < 9)
ACK	01	Acknowledge	
ABO	02	Abort Motion	
STO	03	Stop/Suspend Motion	
STA	04	Startup	
TRA	05	Travel	l/r/i/ l :arc length r :radius of curv. i : 1 = immediate

CHP	06	Change Position	x/y/h/ x : A x pea. y : A y pos. h : A heading
STM	11	Set time to zero	
SVL	12	Set Velocity	•/i/ • : velocity i : 1 *m* immediate
SAC	13	Set Acceleration	a/i/ a : acceleration i : 1 s immediate
SP1	14	Set Pan 1	p/ P : pan angle
STl	15	Set Tilt 1	t/ t : tilt angle
SP2	16	Set Pan 2	p/ P : pan angle
STl	17	Set Tilt 2	t/ t : tilt angle
SSR	18	Set Steering Rate	s/ s : steering rate (0-99 %)
INF	21	Get Vehicle Info	
POS	22	Get position	
TIM	23	Get Vehicle Time	
VH«	24	Get Vehicle Velocity	
ACC	25	Get Vehicle Ace.	
PK1	26	Get Pan Angle 1	
TXil	27	Get TUt Angle 1	
PH2	28	Get Pan Angle 2	
TL1	29	Get TUt Angle 2	
STR	30	Get Steering Rate	
ROXi	31	Get Roll	
KLR	32	Get Roll Rate	
PIT	33	Get Pitch	
PTR	34	Get Pitch Rate	
HBR	35	Get Heading Rate	
REP	36	Get Status	d/ d : device number

Responses from Virtual Vehicle

Mnemonic	Opcode	Maaxiing	Arguments
NAK	00	Negative Ac^moirledge	cc/ cc« 00 : Packet length error 01 : Hw. of Arçs error 02 : Hot ready for and 03 : Illegal Opcode *IX* : *X'th* argument below current maximum limit (1 < *X* < 9) *2X m X'th* argtnyant above current naxiiaus limit (1 < *X* < *9)*
ACK	01	Acknowledge	
KfZ	51	Vehicle In£o	1/w/h/g/a/x/y/s/

			l : length
			w : width
			h : height
			g : weight
			m : minimum turning radius
			x : X of C.G
			y : Y of C.G
			s : 0 = wheel steer
			1 = skid steer
RVP	52	Vehicle position	x/y/h/t/
			x : x position
			y : y position
			h : heading
			t : time
RVT	53	Vehicle Time	t/
RVV	54	Vehicle Vel.	v/t/
RVA	55	Vehicle Acc.	a/t/
RP1	56	Pan angle 1	p/t/
RT1	57	Tilt Angle 1	t/t/
RP2	58	Pan angle 2	p/t/
RT2	59	Tilt Angle 2	t/t/
RSR	60	Steering Rate	s/t/
RRL	61	Roll	r/t/
RRR	62	Roll Rate	r/t/
RPT	63	Pitch	p/t/
RPR	64	Pitch Rate	p/t/
RHR	65	Heading Rate	h/t/
RST	66	Report Status	d/s/t/
			d : device number
			s : status code
ADN	80	Arcdone	id/x/y/h/t/
			id : arc ID code
			x,y : Pos. at end of arc
			h : heading at end of arc
			t : time at end of arc

Note: All distance units are cm, time units are msecs, velocity units are cm/s, acceleration units are cm/s^2 and angle units are half degrees. Weight units are kilograms.

V. References

Goto, Y. and A. Stentz. "The CMU System for Mobile Robot Navigation," in *IEEE International Conference on Robotics and Automation*, 1987.

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

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

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

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

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

Thorpe, C. *1986 Year End Report for Road Following at Carnegie Mellon*, Technical Report, The Robotics Institute, Carnegie Mellon University, 1986.

Wallace, R., A. Stentz, C. Thorpe, C. Moravec, W. Whittaker, and T. Kanade. "First Results in Robot Road-Following," in *International Joint Conference on Artificial Intelligence*, 1985.

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

Section III

Vision and Navigation for the Carnegie Mellon Navlab

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

1. Introduction

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

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

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

2. Navlab: Navigation Laboratory

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

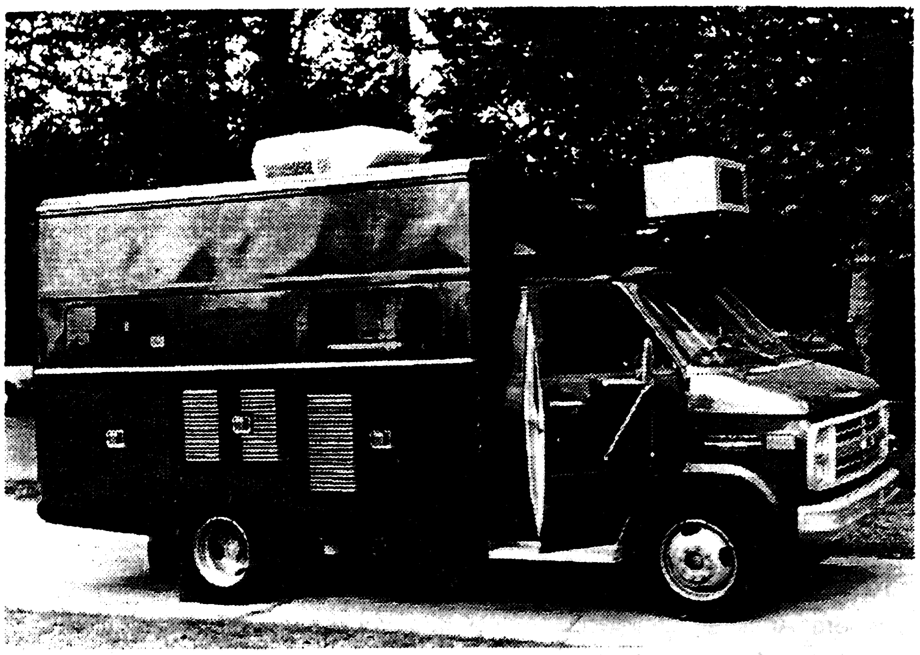


Figure 1: The Navlab

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

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

Features of the Navlab include:

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

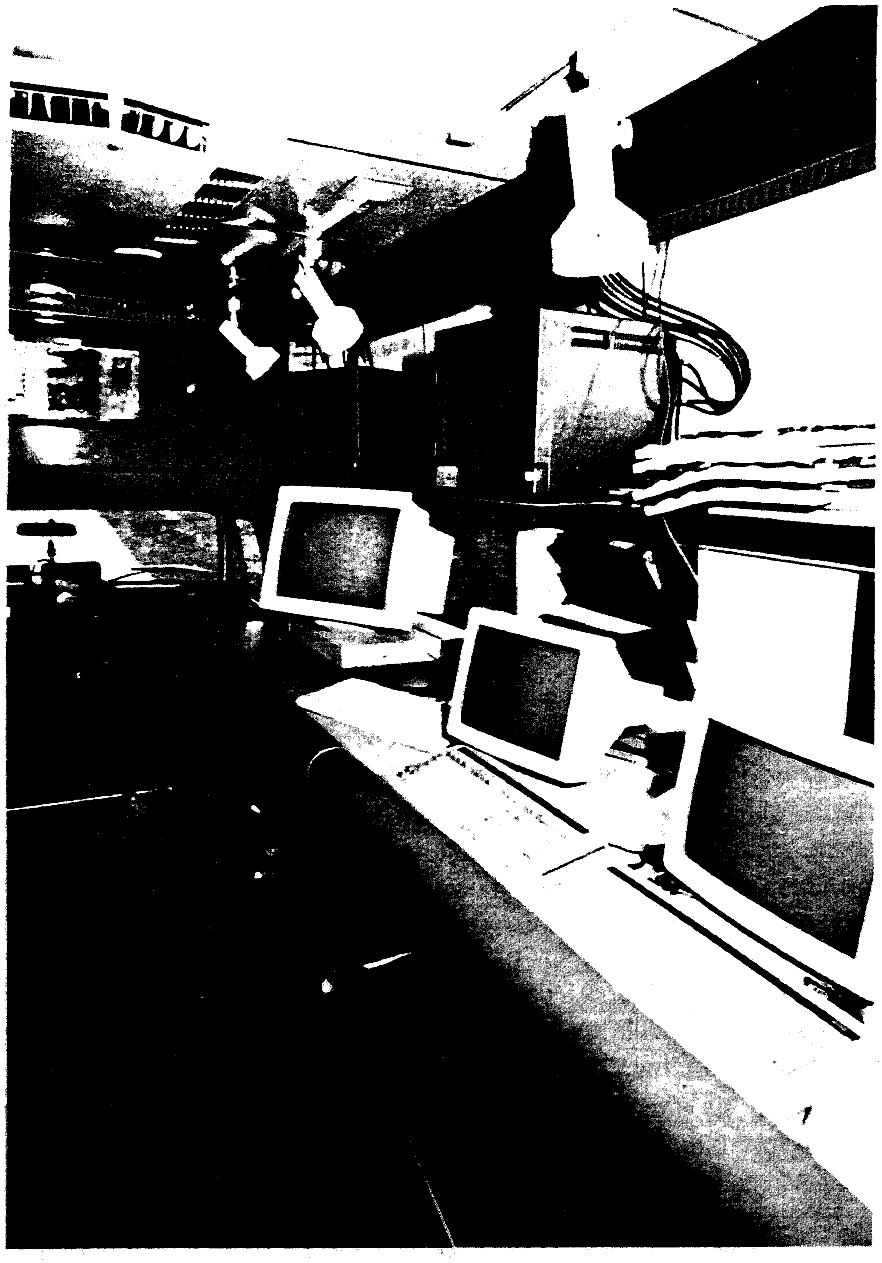


Figure 2: Navlab interior

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

3. Color Vision

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

3.1. Vision Principles for the Real World

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

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

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

Work in the image. The road can be found either by projecting the road shape into the image and searching in image coordinates, or by back projecting the image onto the ground and searching in world coordinates. The problem with the latter approach comes in projecting the image onto an evenly spaced grid in the world. The points on the world grid close to the vehicle correspond to a big area in the lower part of the image; points farther away may correspond to one or a few pixels near the top. Unless one uses a complex weighting scheme, some image pixels (those at the top that project to distant world points) will have more weight than other (lower) points. A few noisy pixels can then have a big or a small effect, depending on where in the image they lie. On the other hand, working directly in the image makes it much easier to weight all pixels evenly. We can directly search for the road shape that has the most road pixels and the fewest non-road pixels. Moreover, projecting a road shape is much more efficient than back projecting all the image pixels.

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

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

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

3.2. Road Following Algorithm

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

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

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

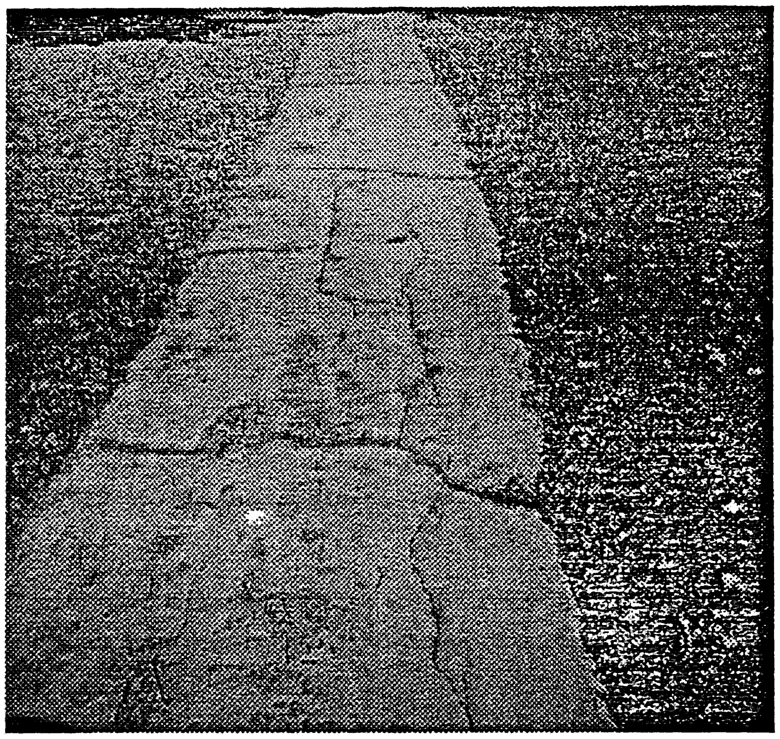


Figure 3: Original Image

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

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

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

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

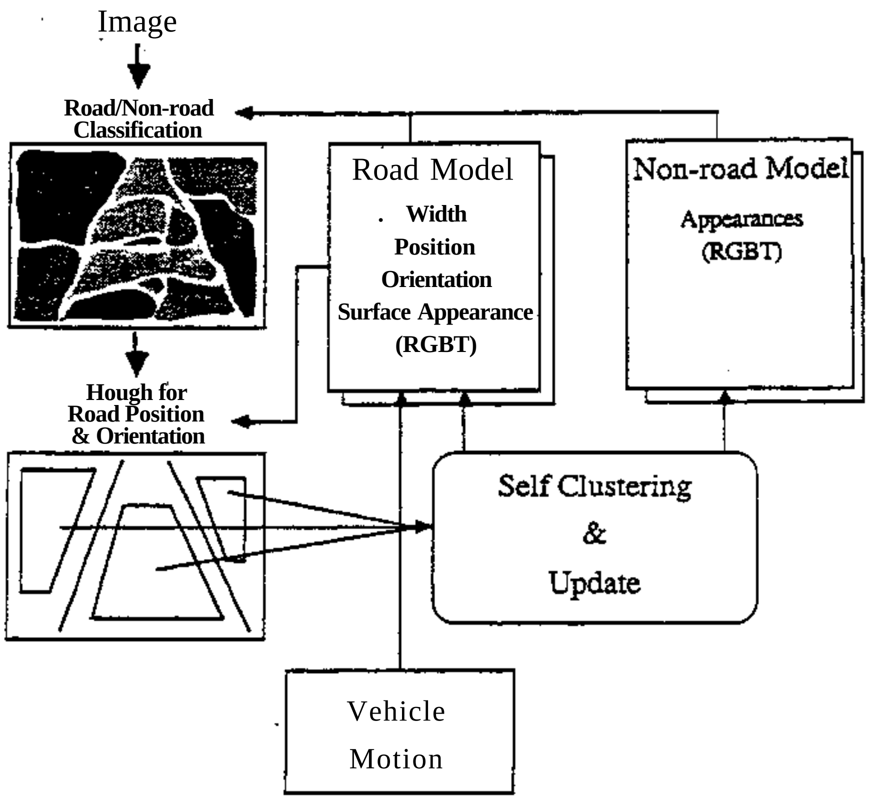


Figure 4: Color vision for road following, including color classification, Hough transform for road region detection, and updating multiple road and non-road models.

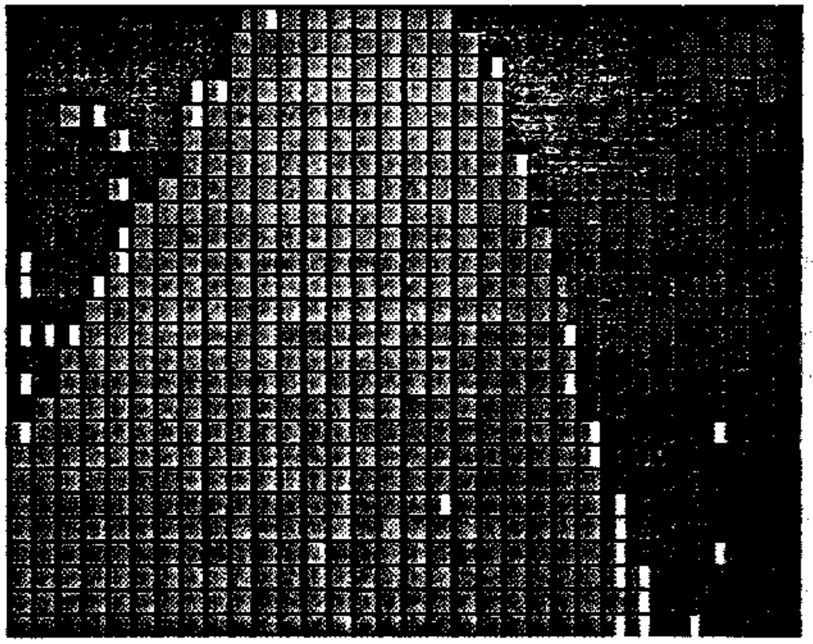


Figure 5: Segmented image. Cotor and texture cues are used to label points below the horizon into two mad arxl two offraad classes

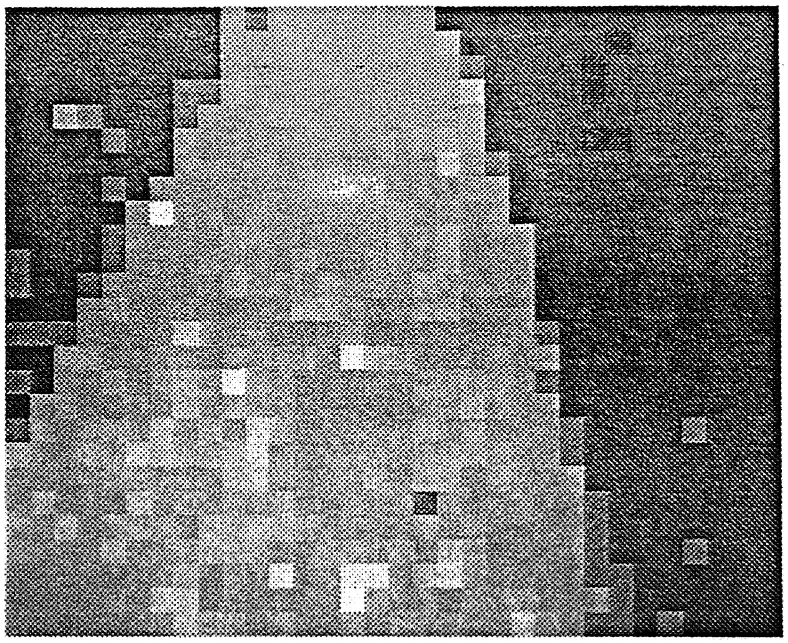


Figure 6: Road probability image. The pixels that best match typical road colors are displayed brightest.

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

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

3.3. Implementation, Details, and Results

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

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

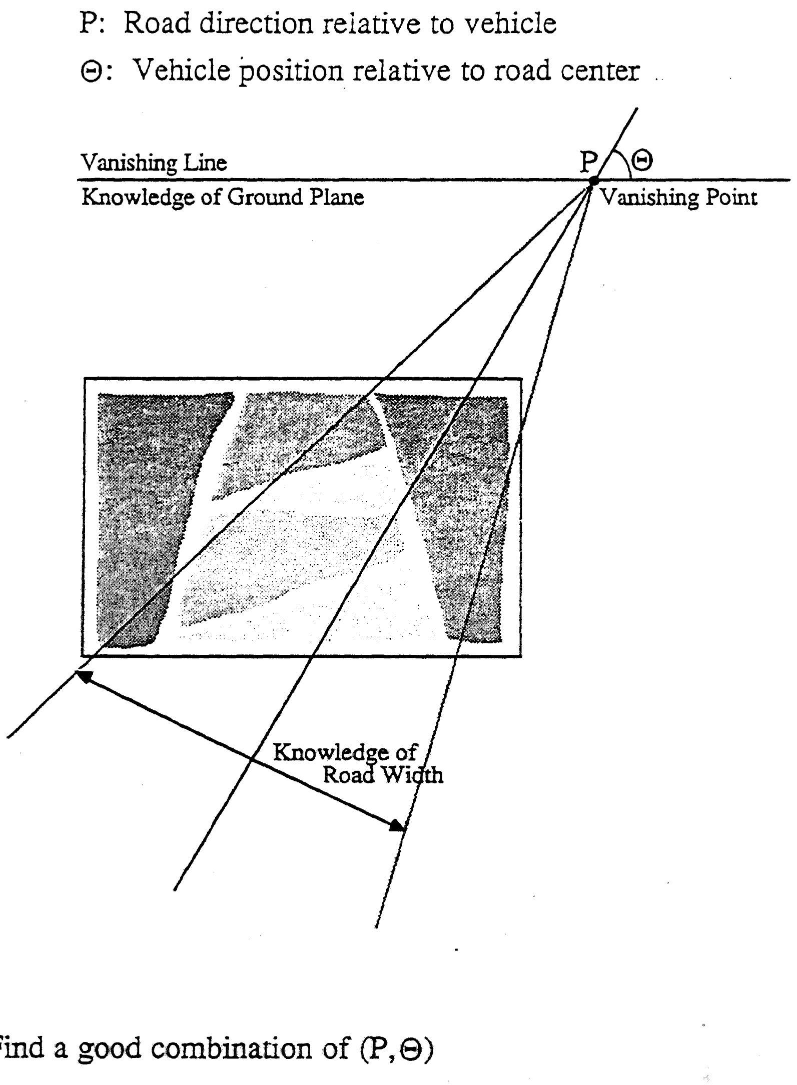

P: Road direction relative to vehicle
Θ: Vehicle position relative to road center
Vanishing Line
Knowledge of Ground Plane
P
Θ
Vanishing Point
Knowledge of Road Width
Find a good combination of (P, Θ)

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

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

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

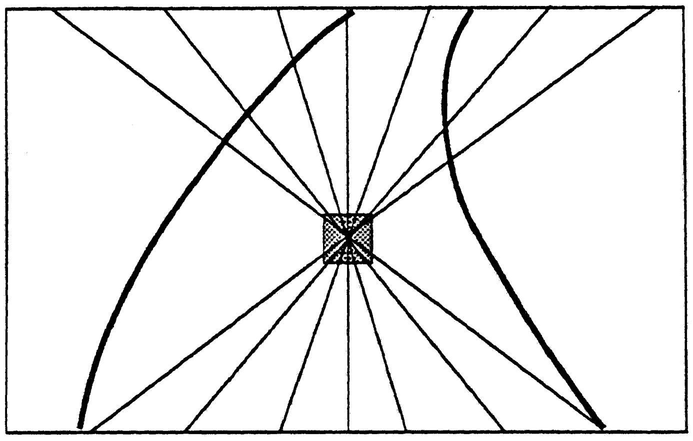


Figure 8: A road point could be a part of roads with different orientations and vanishing points.

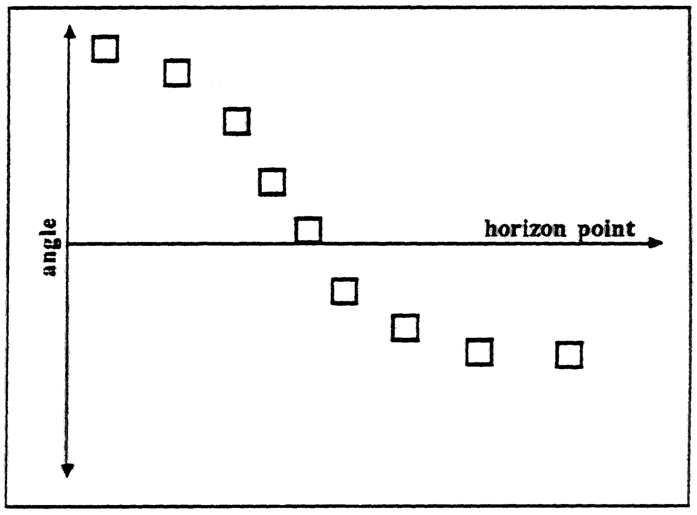


Figure 9: The point from figure 8 would vote for these orientation / intercept values.

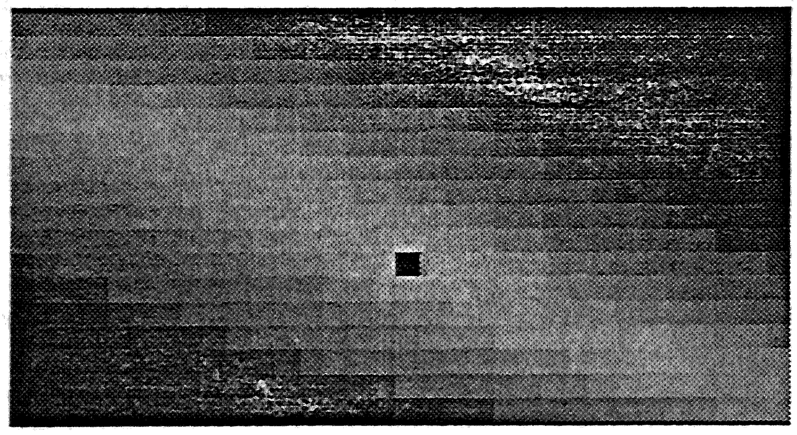


Figure 10: Votes for best road orientation and intercept, and point with most votes (dark square), for road in figure 3.

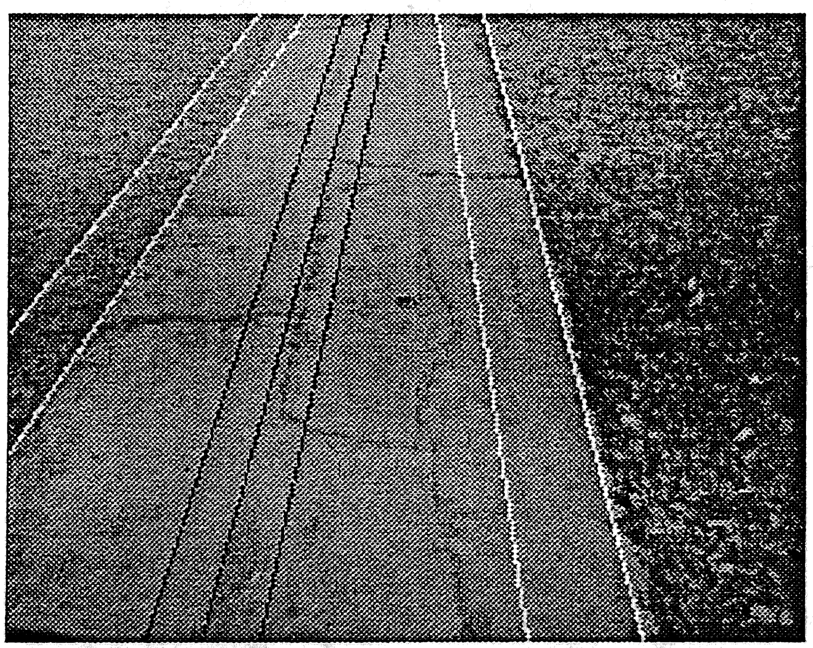


Figure 11: Detected road, from the point with the most votes shown in figure 10.

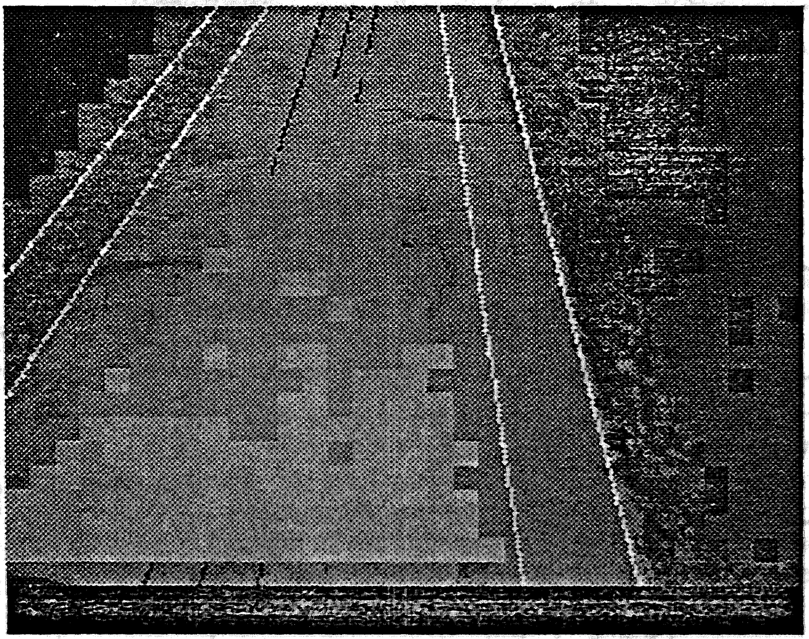


Figure 12: Updating road and nonroad model colors, leaving a safety zone around the detected road region.

Texture Calculation. This is composed of six substeps:

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

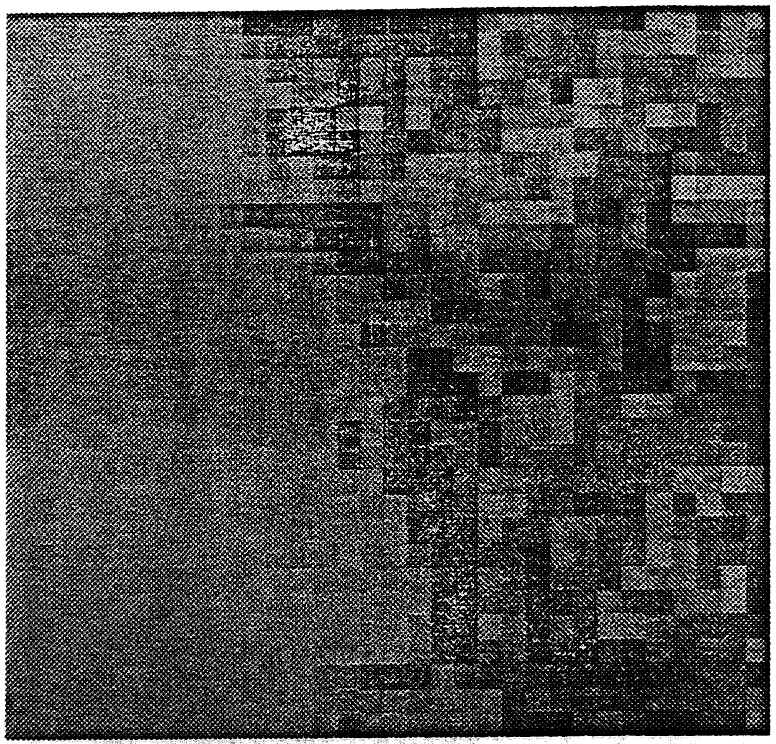


Figure 13: Zoomed picture of road-nonroad boundary. The road (at left) is much less textured than the grass (at right).

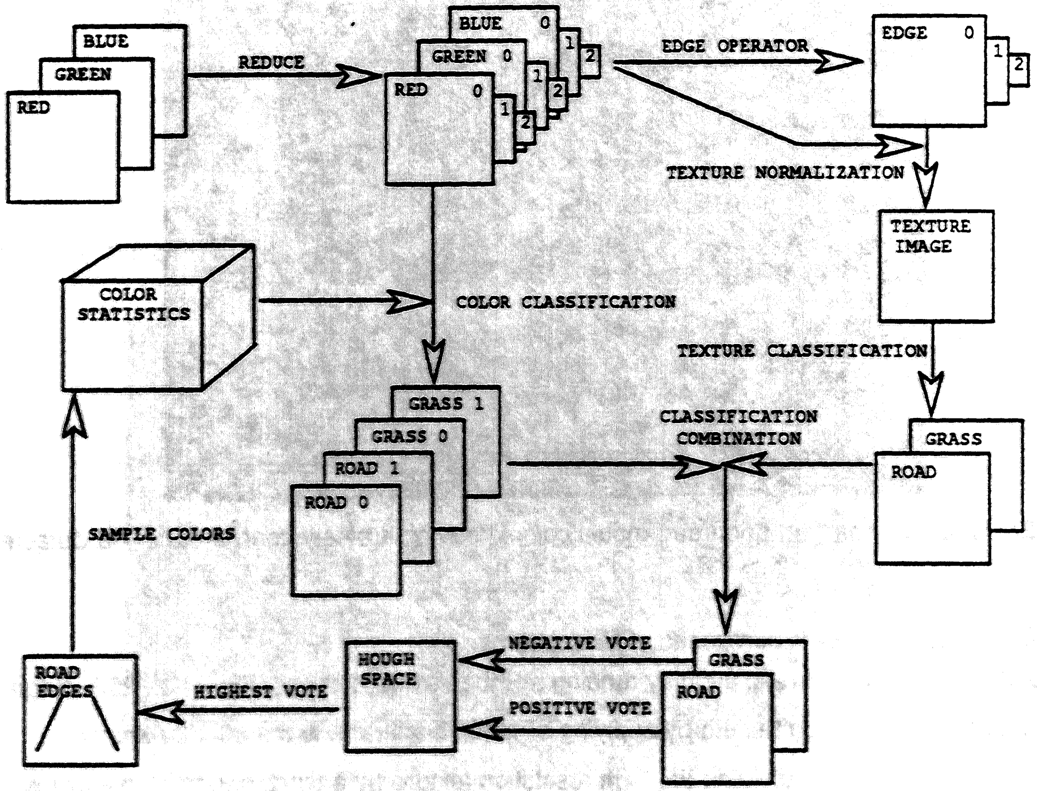


Figure 14: Processing cycle for color vision.

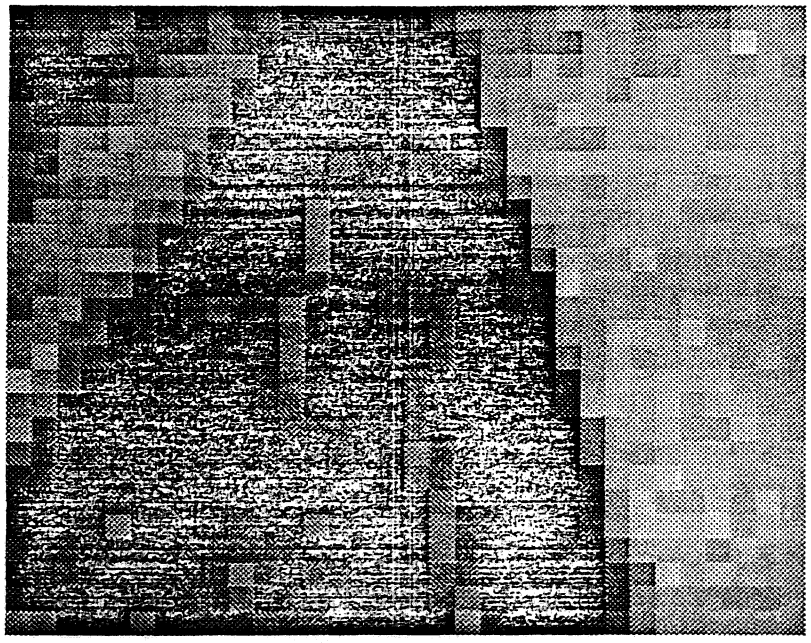


Figure 15: Low resolution texture image, showing textures from figure 3. The brighter blocks are image areas with more visual texture.

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

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

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

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

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

Vote for Best Road Position. This step uses a 2-D parameter space similar to a Hough transform. Parameter 1 is the column of the road's vanishing point, quantized into 32 buckets because the image on which the classification and voting are based has 32 columns. Parameter 2 is the road's angle from vertical in the image, ranging from -1 to 1 radian in 0.1 radian steps. A given road point votes for all possible roads that would contain that point. The locus of possible roads whose centerlines go through that point is an arctangent curve in the parameter space. Because the road has a finite width, the arctan curve has to be widened by the width of the road at that pixel's image row. Road width for a given row is not a constant over all possible road angles but is nearly constant enough that it doesn't justify the expense of the exact calculation. Each pixel's vote is weighted by its calculated confidence. Pixels classified as non-road cast negative votes (with their weights reduced by a factor of 0.2) while road pixels add votes. In pseudo C code, the voting for a pixel at (row, col) is

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

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

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

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

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

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

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

4. Perception in 3-D

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

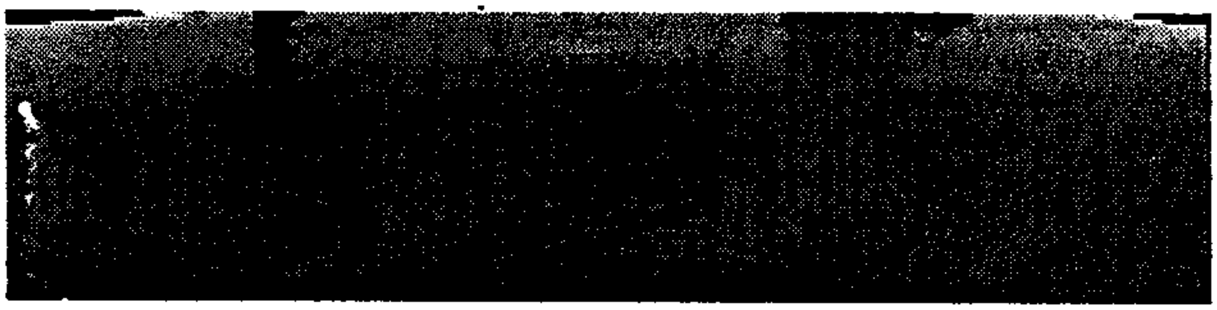


Figure 16: Range image of two trees on flat terrain. Gray levels encode distance; nearer points are painted darker.

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

We have two main processing modes: obstacle detection and terrain analysis. Obstacle detection starts by calculating surface normals from the x-y-z points. Rat, traversaWe surfaces wifl have vertical surface normals. Obstacles wifl have surface patches with normals pointed in other directions. This analysis is relatively fast, running in about 5 s on a Sun 3/75, and is adequate for smooth terrain with discrete obstacles.

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

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

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

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

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

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

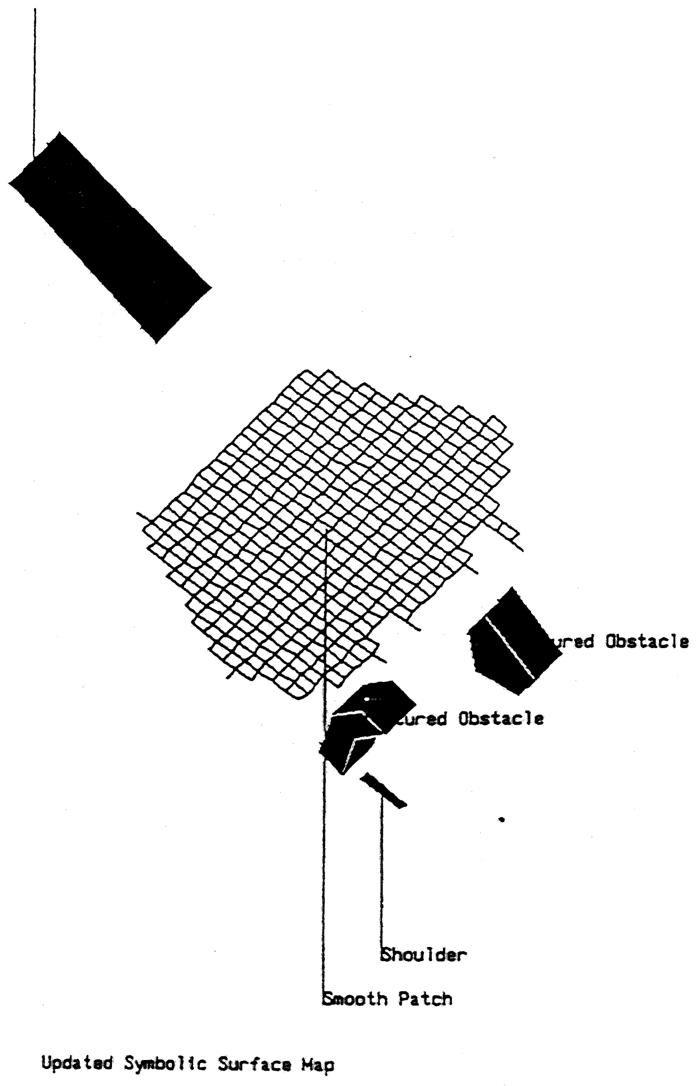


Figure 17: The resultant description of 3D terrain and obstacles from the image in figure 16. The navigable area is shown as a mesh, and the two trees are detected as "textured obstacles" and shown as black polygons

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

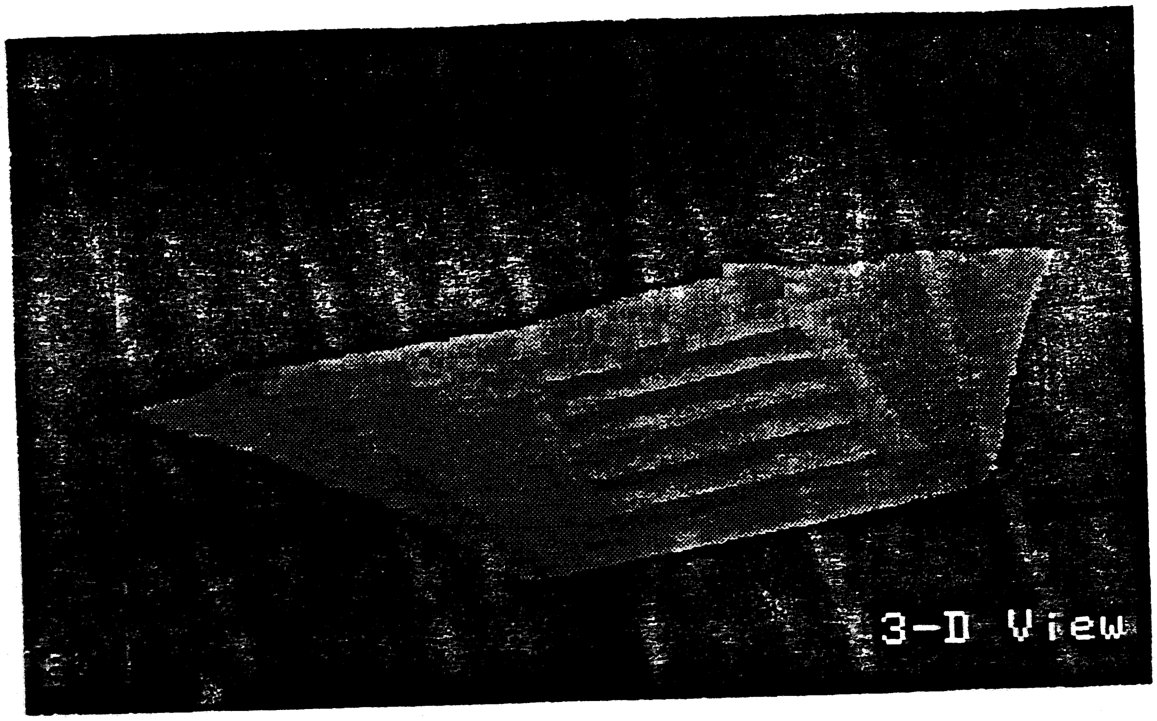


Figure 18: Painted range image of campus stairs. Each point is a {red, green, blue, x, y, z} 6-vector. This image has been rotated and projected from a different viewpoint. The color and range images are registered, so the color edges and regions line up with range edges and regions.

5. System Building

5.1. Artificial Intelligence for Real World Robots

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

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

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

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

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

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

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

Handle real-time vs symbolic interface. At one level, a mobile robot reasons symbolically about perceived objects and planned paths, probably on a slow time scale. At the same time, the vehicle is constantly moving, and low-level servo processes are controlling steering and motion. The top level processes need to be free to take varying amounts of time to process scenes of varying difficulty. They are often event driven, running when a particular object is seen or a particular event occurs. The servo processes, though, must run continuously and in real time (not "simulated real time" or "real time not counting garbage collection^{1*}"). The system should provide for both real-time and asynchronous symbolic processes, and for communications between them.

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

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

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

Do not throw away geometric precision. Mobile robots need all the information they can get. It is often important to know as precisely as possible where an object is located, either for planning efficient paths or for updating vehicle location. There is TO need to turn a measured distance of 3.1 m into *fairiy* *close*. Given the relative costs and speeds of computers and vehicles, it is more efficient to spend extra computing effort (if any) to handle precise data than to plan fuzzy paths that take the vehicle unnecessarily far out of its way.

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

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

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

5.2. Blackboards and Whiteboards

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

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

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

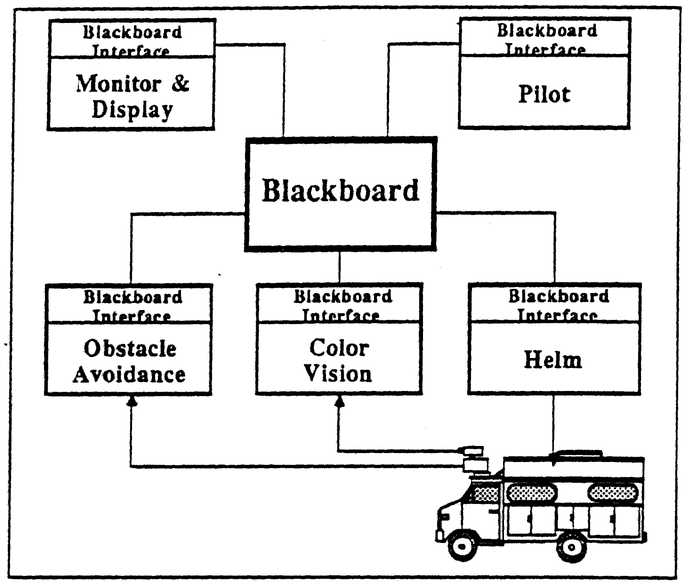


Figure 19: Navlab software architecture

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

5.3. Data Storage and Retrieval

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

A module can store a token by calling a subroutine to send it to the LMB. Tokens can be retrieved by constructing a pattern called a *specification* and calling a routine to request that the LMB send back tokens matching that specification. The specification is simply a Boolean expression in which the attributes of each token may be substituted; if a token's attributes satisfy the Boolean expression, then the token is sent to the module that made the request. For example, a module may specify:

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

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

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

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

or

tokens with **location** *overlapping X*

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

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

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

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

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

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

5.4. Synchronization Primitives

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

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

5.5. Architecture

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

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

perceived road.

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

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

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

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

6. Conclusions and Future Work

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

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

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

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

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

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

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

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

Acknowledgments

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

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

References

[Cohen 82]	Cohen, P., Barr, A., Feigenbaum, E., eds. *The Handbook of Artificial Intelligence.* William Kaufman, 1982.
[Goto 86]	Goto, Y., Matsuzaki, K., Kweon, I., Obatake, T. CMU sidewalk navigation system. In *Fall Joint Computer Conference.* ACM/IEEE, 1986.
[Hebert 86]	Hebert, M., Kanade, T. Outdoor scene analysis using range data. In *IEEE International Conference on Robotics and Automation.* 1986.
[Rychener 86]	Rychener, M. D., Farinacci, M. L., Hulthage, I., Fox, M. S. Integration of multiple knowledge sources in Alladin, an alloy design system. In *AAAI-1986.* AAAI, 1986.
[Shafer 86]	Shafer, S., Stentz, A., Thorpe, C. An architecture for sensor fusion in a mobile robot. In *IEEE International Conference on Robotics and Automation.* 1986.
[Singh 86]	Singh, J., et al. *NavLab: an autonomous vehicle.* Technical Report, Carnegie Mellon Robotics Institute, 1986.
[Thorpe 86]	Thorpe, C. Vision and navigation for the CMU Navlab. In *SPIE.* Society of Photo-Optical Instrumentation Engineers, October, 1986.

Section IV

The CMU System for Mobile Robot Navigation

Yoshimasa Goto
Anthony Stentz
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

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

1. Introduction

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

1.1. The Test Sites and Vehicles

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

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

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

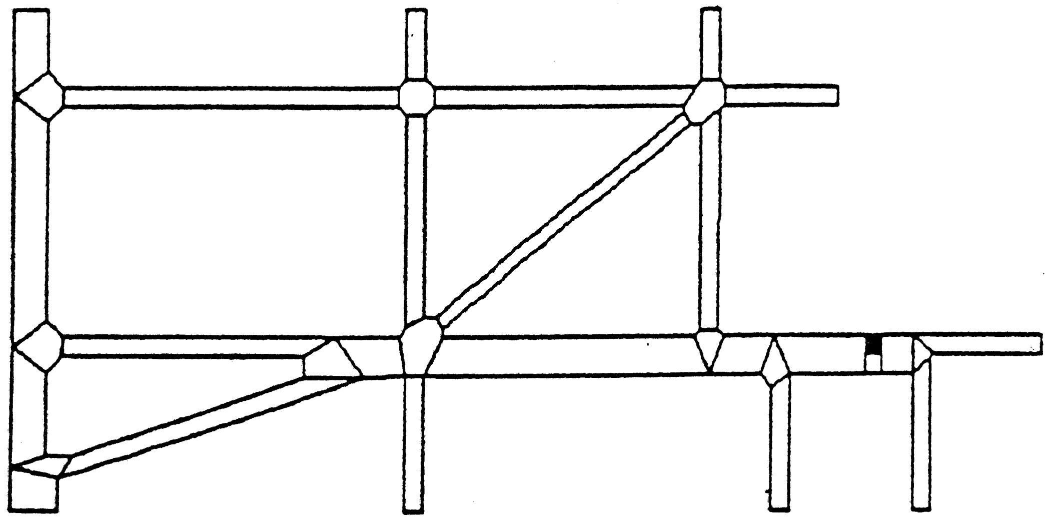


Figure 1: Map of the CMU Campus Test Site

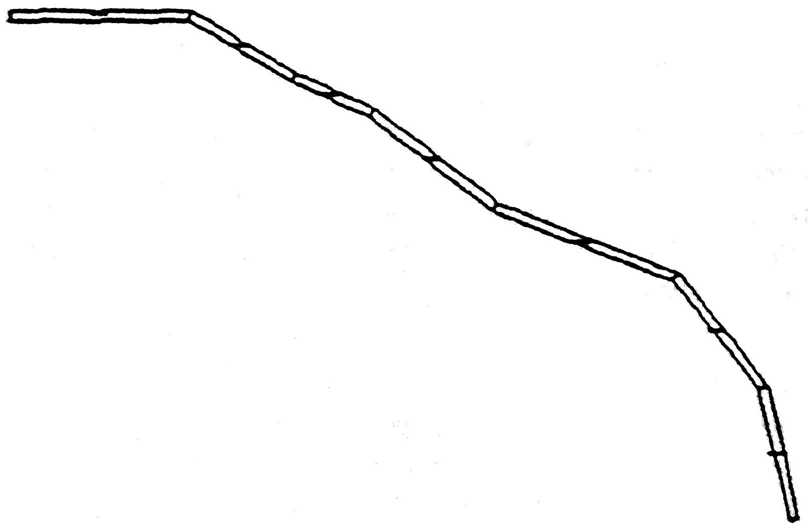


Figure 2: Map of the Schenley Park Test Site

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

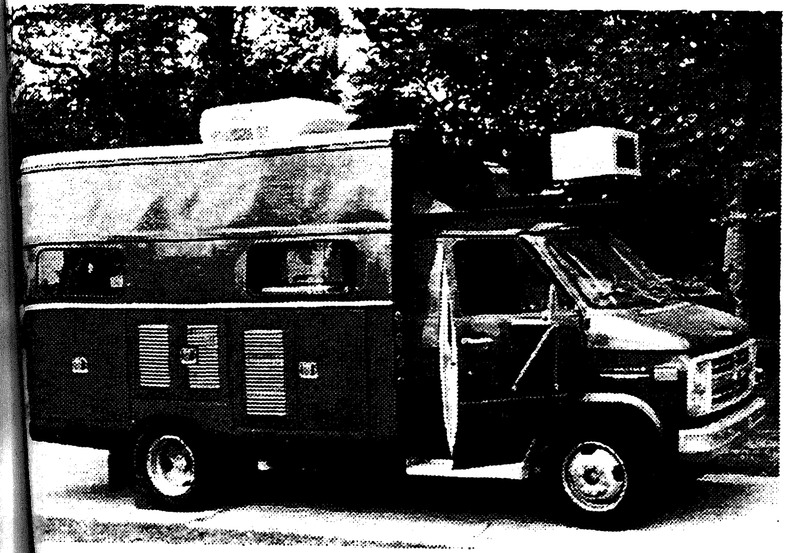
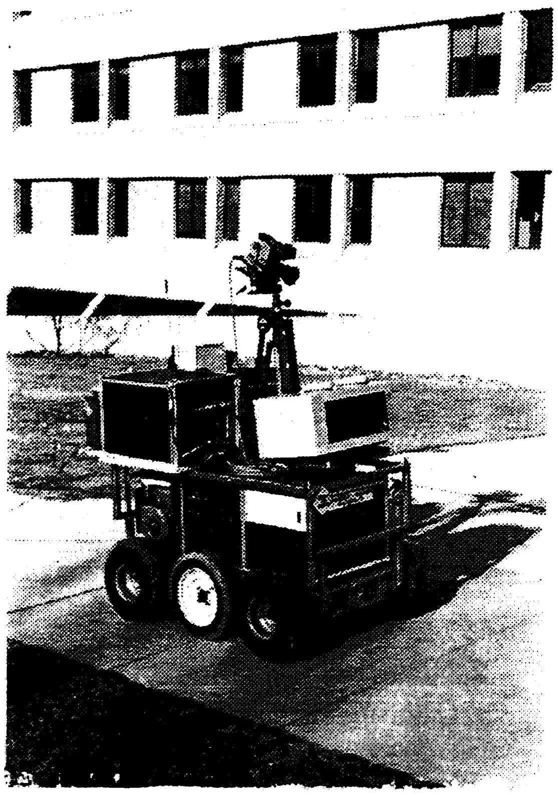



Figure 3: The Navlab and Terregator

1.2. Current System Capabilities

Currently, the system has the capability.

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

2. Design of the System Architecture

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

2.1. Design Goals and Principles

The goals of our outdoor navigation system are:

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

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

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

2.2. Outdoor Navigation Tasks

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

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

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

2.3. System Architecture

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

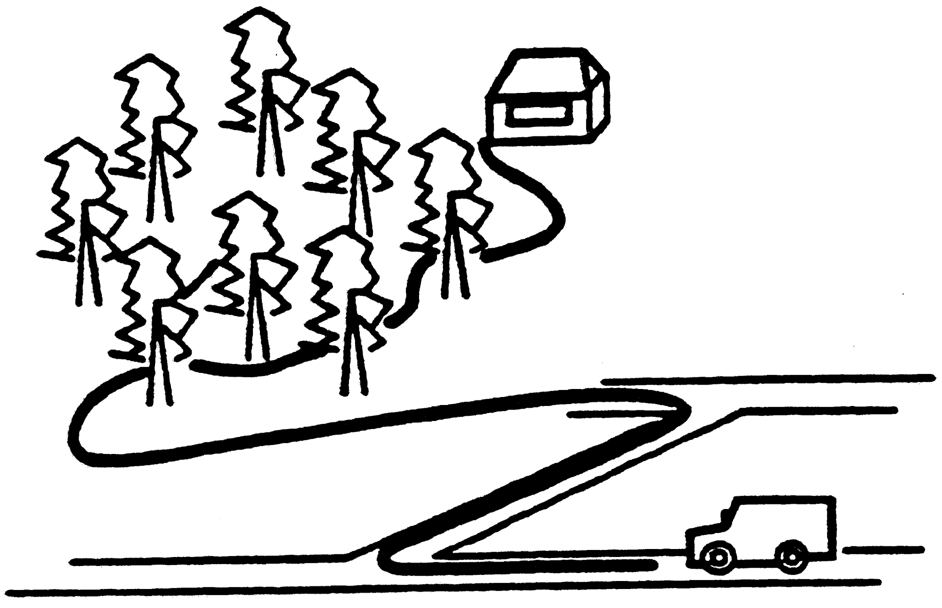


Figure 4: Outdoor navigation

2.3.1. Module Structure

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

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

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

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

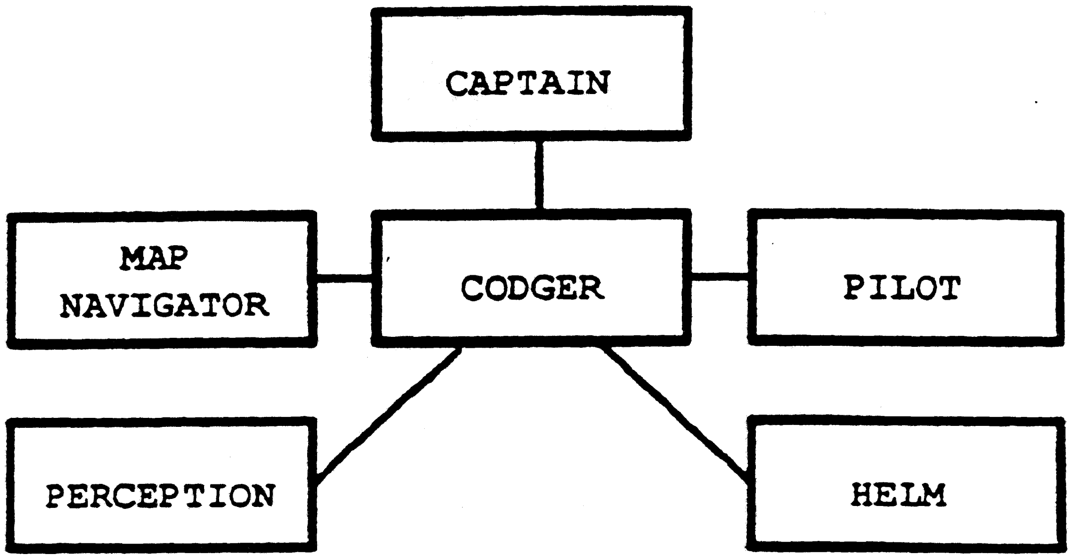


Figure 5: System architecture

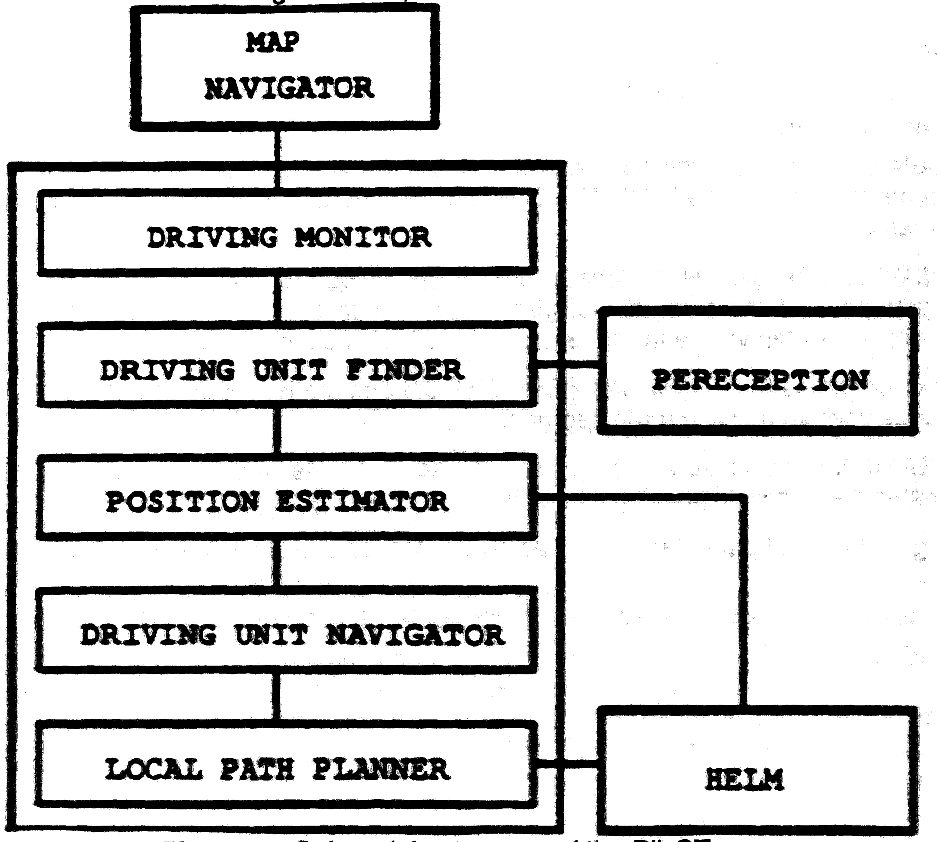


Figure 6: Submodule structure of the PILOT

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

2.3.2. CODGER

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

3. Parallelism

3.1. The CODGER System for Parallel Processing

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

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

3.1.1. Data Representation

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

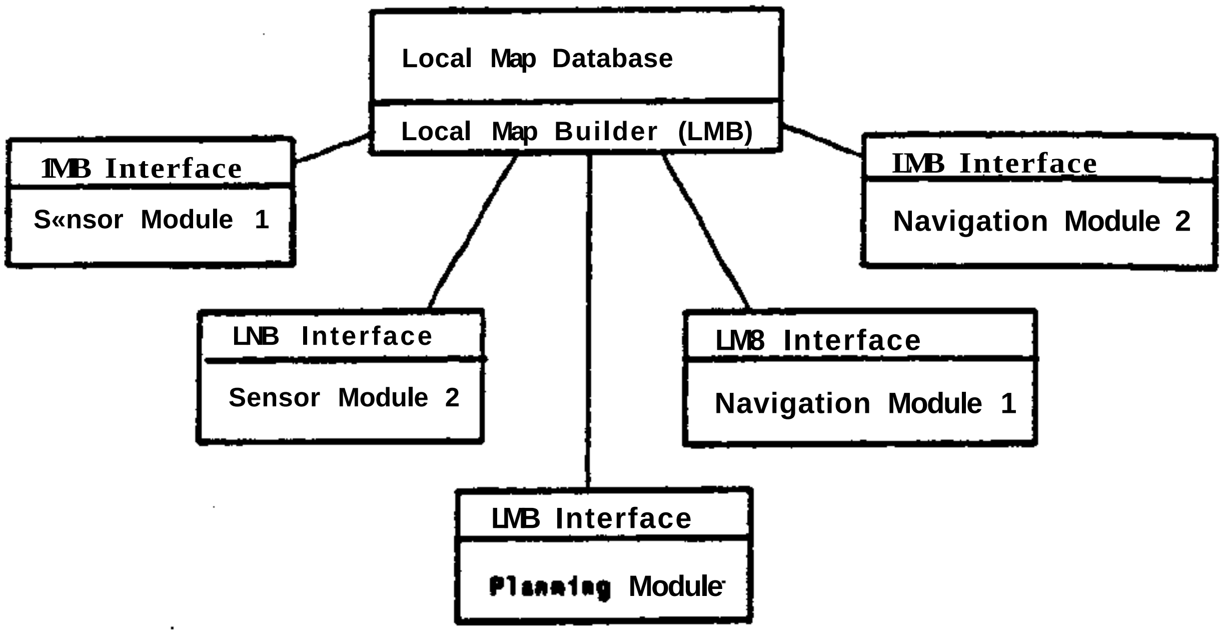


Figure 7: The CODGER software system

3.1.2. **Synchronization**

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

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

3.2. Parallel Asynchronous Execution of Modules

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

3.2.1. Global and Local Navigation

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

3.2.2. Driving Pipeline

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

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

4. Sensor Fusion

4.1. Types of Sensor Fusion

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

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

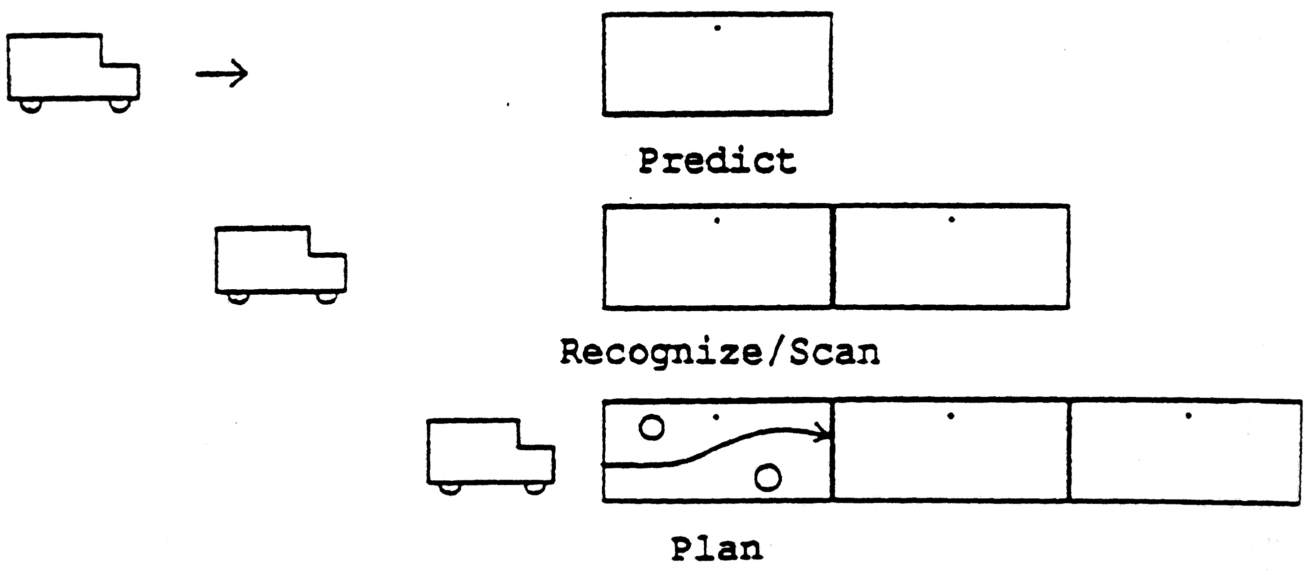


Figure 8: Driving pipeline

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

- **Complementary:** Sensors provide data of different modalities. The task of recognizing three-dimensional objects illustrates this kind of fusion. In the CMU systems, a set of stairs is recognized using a color camera and laser rangefinder. The color camera provides *image* information (e.g., color and texture) while the laser rangefinder provides *three-dimensional* information.

- **Independent:** A single sensor is used for each task. An example of a task requiring a single sensor is distant landmark recognition. In this case, only the camera is used for landmarks beyond the range of the laser rangefinder.

4.2. Examples of Sensor Fusion Tasks

4.2.1. Vehicle Position Estimation

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

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

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

4.2.2. Pilot Control

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

4.2.3. Colored Range Image

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

4.3. Problems and Future Work

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

5. Local Control

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

5.1. Adaptive Driving Units and Sensor View Frames

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

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

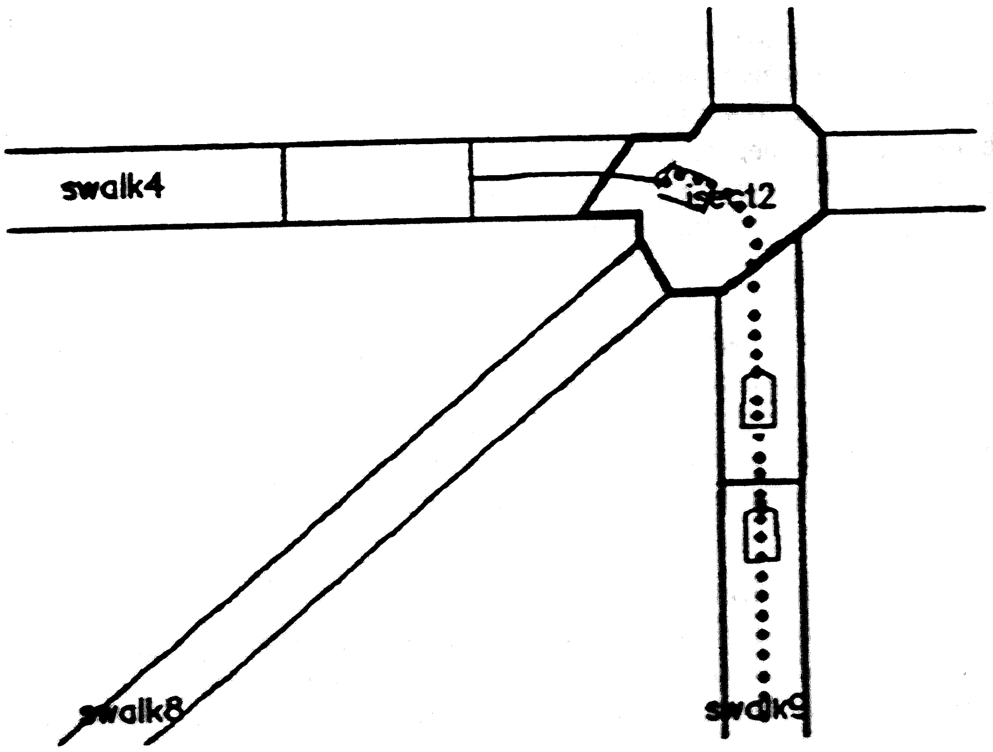


Figure 9: Intersection driving unit

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

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

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

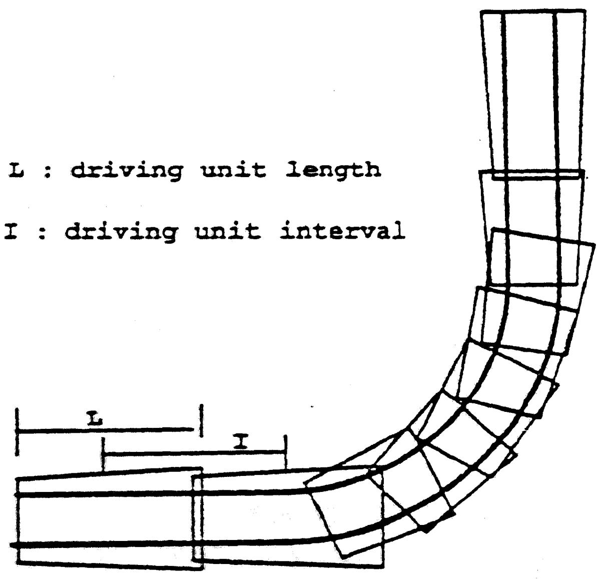


Figure 10: Adaptive Driving Units

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

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

5.2. Vehicle Speed

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

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

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

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

5.3. Local Path Planning and Obstacle Avoidance

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

6. Navigation Map

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

6.1. Map Structure

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

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

6.2. Data retrieval

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

7. Other Tasks of the System

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

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

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

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

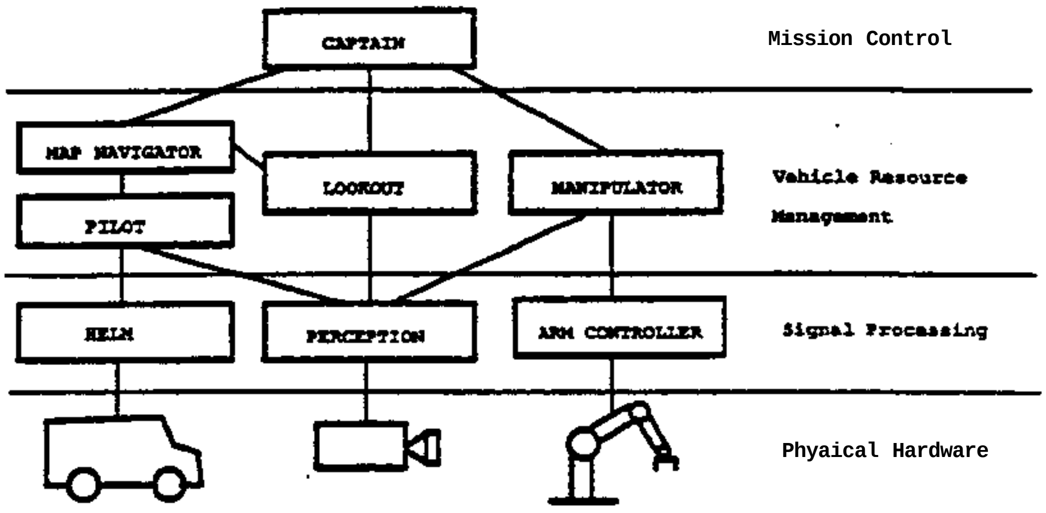


Figure 11: Extended system architecture

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

8. Conclusions

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

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

We have demonstrated our system's afalfty to drive around a network of sidewalks and along a curved road* recognize complicated landmarks, and avoid obstacles. Future work will foots on improving CODGER for handing *mom* difioit sensor fusion problems. We wil also work on better schemes for local navigation and wil strive to reduce oyr dependence on map data.

9. Acknowledgements

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

References

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

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

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

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

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

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

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

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

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

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

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