Development of an Integrated Mobile Robot System at Carnegie Mellon University: December 1989 Final Report

Steve Shafer and William Whittaker

CMU-RI-TR-90-12

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

January 1990

This research was 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.

Contents

Abstract

This report describes progress in development of an integrated mobile robot system at the Robotics Institute of Carnegie Mellon University from July 1988 to December 1989. This research was sponsored by the Defense Advanced Research Projects Agency and monitored by the U.S. Army Engineer Topographic Laboratories under contract DACA76-86-C-0019.

In this program, we pursued a broad agenda of research in the development of mobile robot vehicles, focused on the *NAVLAB* computer-controlled van. In the period covered by this report, July 1988 to December 1989, we addressed major software issues for mobile robot vehicles:

- **Evolution of the CODGER Blackboard and the Driving Pipeline Architecture.**
  - CODGER is the blackboard system for the NAVLAB that synchronizes and passes data among the various processing modules. The Driving Pipeline is a set of modules that operate in parallel to implement continuous motion control, road-following, and obstacle avoidance. In this reporting period, our extensions to CODGER and the Driving Pipeline include adaptive adjustment of planning parameters to give desired robot responsiveness and control parameters to ensure smooth operation.
- **Kinematic Path Planning for Wheeled Vehicles.**
  - We developed a new method for vehicle path planning designed to handle off-road scenarios rather than traditional "flat-world" scenarios. In the past, most path planning algorithms have assumed that the entire world is flat, with polygonal obstacles clearly identified. However, the terrain encountered in cross-country driving presents more subtle problems in vehicle tilt and clearance; these interact with vehicle constraints such as minimum turning radius. Our new path planner can explicitly model and account for such aspects of the problem, and has been optimized to work fast enough for use on the NAVLAB driving cross-country.

This software is central to the **New Generation System** (NGS) for robot vision and navigation, which combines many independent technologies to produce an integrated mobile robot system.

Acknowledgements

This research has been a team effort involving many people, including: Steve Shafer, William Whittaker, Takeo Kanade, Tony Stentz, Chuck Thorpe, Paul Allen, Gary Baun, Mike Blackwell, Kevin Dowling, Thad Druffel, James Frazier, Taka Fujimori, Yoshi Goto, Eric Hoffman, Ralph Hyre, Inso Kweon, James Ladd, James Martin, Clark McDonald, Jim Moody, Henning Pangels, David Simon, Bryon Smith, and Eddie Wyatt.

Section I
Introduction

Introduction and Overview

This report describes progress in development of an integrated mobile robot system at the Robotics Institute of Carnegie Mellon University from July 1988 to December 1989. This research was sponsored by the Defense Advanced Research Projects Agency and monitored by the U.S. Army Engineer Topographic Laboratories under contract DACA76-86-C-0019.

In this program, we pursued a broad agenda of research in the development of mobile robot vehicles, focused on the *NAVLAB* computer-controlled van. In the period covered by this report, July 1988 to December 1989, we addressed major software issues for mobile robot vehicles:

- **Evolution of the CODGER Blackboard and the Driving Pipeline Architecture.**
  - CODGER is the blackboard system for mobile robots designed to handle top-down, map-based navigation on roads: it uses a general map format with semantic and geometric levels defining perceivability and navigability, and supports both global and local navigation.
  - The Local Navigator, a set of modules that interact through CODGER, handles parallelism and synchronization in sequencing vehicle operations. These modules implement a control paradigm we call the Driving Pipeline, in which different modules look at the road at different distances in front of the vehicle.
  - In this reporting period, we added extensions to this architecture to automatically select driving and scanning distances to maximize parallelism, adaptively adjust planning parameters to give desired robot responsiveness, and adaptively adjust control parameters for smooth operation.
- **Kinematic Path Planning for Wheeled Vehicles.**
  - In cross-country driving, the primary problem is not the use of parallelism for fast motion, but rather is how to find pathways that are safe to navigate without fear of harming the vehicle or getting stuck; yet are aggressive enough to allow the vehicle to navigate even through tight spaces.
  - To accomplish this, we developed a new method for vehicle path planning designed to handle off-road scenarios rather than traditional "flat-world" scenarios. One aspect of this path planner is modeling the properties of the terrain that would prevent the vehicle from passing safely. In principle, our system can model any mathematical constraints defined across the terrain. Currently, we model the basic terrain restrictions of the vehicle: tilt, locomotion support, and body clearance.
  - In addition, there are limitations on the possible motion of the vehicle. This is important because most robot vehicles, including car-like robots such as the NAVLAB, are not omnidirectional. Our new path planner accounts for this by modeling kinematic constraints as well, such as the minimum turning radius of the vehicle.
  - The path planner must be used by the vehicle very frequently to account for changes in its position and viewing area. Unfortunately, a comprehensive path planner such as ours can be very expensive to run. Therefore, we have invested a great deal of attention in developing representations and algorithms that allow the path planner to run quickly. The most important development is the use of an oct-tree in the configuration space of the vehicle that indicates what positions and orientations are allowed by the terrain constraints. The vehicle kinematics are pre-compiled into oct-tree relationships that allow very fast execution of the path planner while the vehicle is moving.

This software is central to the **New Generation System** (NGS) for robot vision and navigation, which combines many independent technologies to produce an integrated mobile robot system.

Accomplishments

Our accomplishments in this reporting period include:

- Development of the configuration-space approach for kinematic path planning.
- Implementation of the path planner with an oct-tree representation for planning constraints.

Future Research Directions

This is the final report for the current contract, so it marks the termination of the current research program. However, we have identified several topics that we believe are important areas for further research in the general area of mobile robot vehicles:

- Development of a loosely synchronized version of the Driving Pipeline that does not require all sensors to process data at the same rate.
- Optimizing the communications among a set of low-level navigation modules -- obstacle detection, path planning, and progress monitoring -- that form a complete subsystem for vehicle control. This would involve removing them from the general CODGER communication structure and providing specialized communication paths.
- Incorporation of our new kinematic path planner into the NAVLAB navigation architecture.

Section II
Evolution of the CODGER Blackboard and the Driving Pipeline Architecture

Introduction

The CODGER Blackboard System

Most sophisticated mobile robot systems are large and complex [26, 39, 51]. For this reason, such systems are usually developed by a team of researchers, rather than an individual. Furthermore, in order for these systems to execute in a reasonable amount of time, the computations must be parallelized to some degree. These two characteristics pose two problems:

- What software engineering tools are needed for developing a large mobile robot system?
- What software support is needed for successfully distributing the computation across a number of processors?

In the case of the NAVLAB and its development laboratory, the software system consists of five to ten large sensing, planning, control, and graphics modules running on a mixture of Suns and Vaxes interconnected via an EtherNet. Both Lisp and C are available for programming. From a software engineering standpoint, we would like to support module development in multiple languages running on multiple machine architectures, all tied together in a single system. Furthermore, we would like one module programmer to be able to modify or extend one part of the system without requiring redesign or recompilation of the rest of the system. Concerning multiprocessing support, we need a common data format to all modules, regardless of programming language or machine type. We need a mechanism for moving data between modules, and more importantly, synchronizing this exchange. For geometric data, which is used by nearly every module in the system, we must guarantee that it is consistent, so that all modules agree as to the location of the robot and all objects in its environment. A number of systems exist for integrating various modules into a single system and coordinating their activities. Many of these systems are grouped together under the name of *blackboards* [16, 41, 42]. While these systems do permit many modules (or *knowledge sources*) to be integrated into a single system, traditionally multiprocessing is not supported as modules are invoked one at a time. Furthermore, modules are scheduled for execution according to a *static* priority scheme. This mechanism is too restrictive for the NAVLAB system. As illustrated in this section, a *dynamic* scheme is needed that changes from one context to another. Other systems [2, 19] do provide adequate multiprocessing support, but fall short in supporting geometric data, which is central to robotic systems. We have developed a system named CODGER (for COmmunications Database with GEometric Reasoning) that provides ample support for module development and multiprocessing support. In this section we describe CODGER and illustrate how it addresses these issues.

The Driving Pipeline Architecture

Outdoor navigation is a broad and rich problem. In order to devise a workable architecture, the scope of the problem must be narrowed somewhat. Given the NAVLAB mobile robot equipped with a color camera and an ERIM laser rangefinder, we sought to develop an architecture for driving the NAVLAB continuously on- and off-road, using camera-based sensing for road-following and landmark recognition and ERIM-based sensing for obstacle avoidance. We wanted the system to be flexible enough to use map information if provided, and to construct a map if not. We wanted to be able to run the system on

general purpose computers, to allow flexibility in design and ease in development. In order to meet this last objective, we were required to remove all real-time constraints from consideration. Thus, we required that our environment be static (i.e., no moving objects other than the robot), and that no minimum speed be necessary.

A number of issues come to bear on the problem:

- Whether the robot starts with a map or constructs one as it moves, a map *format* is needed for representing roads and other navigable passages, landmarks, obstacles, and other items of interest that can be understood by perceptual and planning modules.
- A suitable division is needed in the architecture between global and local navigation. Global navigation is the task of planning coarse paths for the robot based on map information and overseeing the execution of such paths. Local navigation is the task of coordinating sensing, planning, and control in such a way as to realize the global path incrementally, taking into account features of the local environment too dense to represent in a global map.
- A global planning algorithm is needed to construct coarse paths based on map information, even in the event that the map information is incomplete.
- Local sequencing of sensing, planning, and control must guarantee that the robot does not drive into unexplored areas, thus running the risk of collision.
- The robot must take into account the fact that certain scenarios require more processing than others, and thus the robot's speed must be adjusted to avoid degenerative start/stop motion.
- In order to make effective use of a multiprocessing environment, the entire system must be parallelized as much as possible.

A number of mobile robot architectures have been developed for indoor and outdoor navigation. The first complete system was Shakey [43]. Shakey was an indoor mobile robot equipped with a camera and rangefinder that navigated around polyhedral objects on a flat floor. The Shakey architecture consisted of a resolution-based problem solver for planning tasks and a set of primitive action routines (such as for navigation and manipulation) to carry out these tasks. HILARE [10, 21] was an indoor robot equipped with a camera, rangefinder, and acoustic sensors for proximity sensing. It employed a two-tiered map representing navigation boundaries for a floor plan. The top tier represented just the topology of the floor plan while the bottom tier represented the topographical information using convex polygons. HILARE was able to fill in or update topographical information as it navigated. A number of other systems used acoustical sensors to either build a map of the robots environment [14, 15] or to navigate based on a previously built map [11]. Other systems focused on more difficulty sensing, planning, and tracking problems for mobile robots [25, 40, 50] out of the context of a large system. As mobile robots moved outdoors, the increased difficult of the environment mandated multiple sensors and modes of navigation. The architectures increased in complexity correspondingly [39, 51]. The most advanced of the above systems are similar in many ways. All use statistical pattern recognition or edge-tracing techniques for following roads or typing terrain, and laser rangefinder or sonar sensors for obstacle avoidance. Many plan routes based on a map.

There are some basic differences that set the NAVLAB system architecture [23, 46] apart from the others. The first centers on the map. The above systems used simple maps that represented only the boundaries separating navigable areas from unnavigable areas. The NAVLAB map represents semantic and perceivable geometry along with the navigable geometry. This representation makes explicit the information necessary to plan routes and sensing operations respectively. The system is able to handle incomplete map information and still follow a route plan. The second difference centers on the local

sequencing of operations. The above systems either use stop-and-go motion or drive at a fixed speed, thus placing strict real-time constraints on local processing in order to move continuously. Stop-and-go motion is undesirable because it results in a suboptimal vehicle speed. Strict real-time constraints are difficult to abide by since processing requirements change from environment to environment. The NAVLAB system drives continuously but is able to adjust its speed and even stop to avoid catastrophe in the event that processing bogs down, thus precluding the need to address real-time constraints.

In this section we present an overview of the NAVLAB architecture for navigation and then focus on the local navigation portion of the system. Most of the ideas presented here were implemented and tested in a number of systems [22, 23, 24]. Some of the ideas pertaining to perceivable and navigable geometry in the map and the interface to the local path planner were developed in the NGS system, but were not implemented.

Overview of the CODGER System

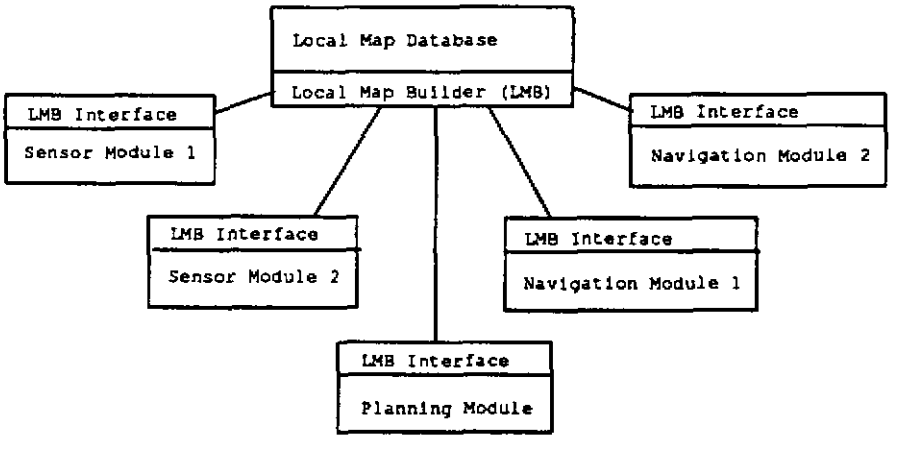


Figure 1: CODGER System Structure

The computing resources for the NAVLAB consist of three to five Sun microcomputers interconnected with an EtherNet. The CODGER system was designed to facilitate communication between system modules for a static mapping of modules to processors in this arrangement. The basic structure of CODGER is illustrated in Figure 1. CODGER consists of a central database, called the *local map*, a program to manage the database, called the *local map builder* (LMB), and a library of subroutines for communicating with the LMB, called the *LMB Interface*. The other boxes in the figure are user modules. These modules run as separate programs in the system. Each module is linked with an LMB Interface, through which the module can store and retrieve information from the database. The LMB Interface handles all communication and synchronization with the LMB over the network.

Note that the system configuration is "star-shaped", that is, all data passed between modules must pass through the central database. There are several advantages to this arrangement. First, during development of the system, the communication paths between modules are frequently changed as the

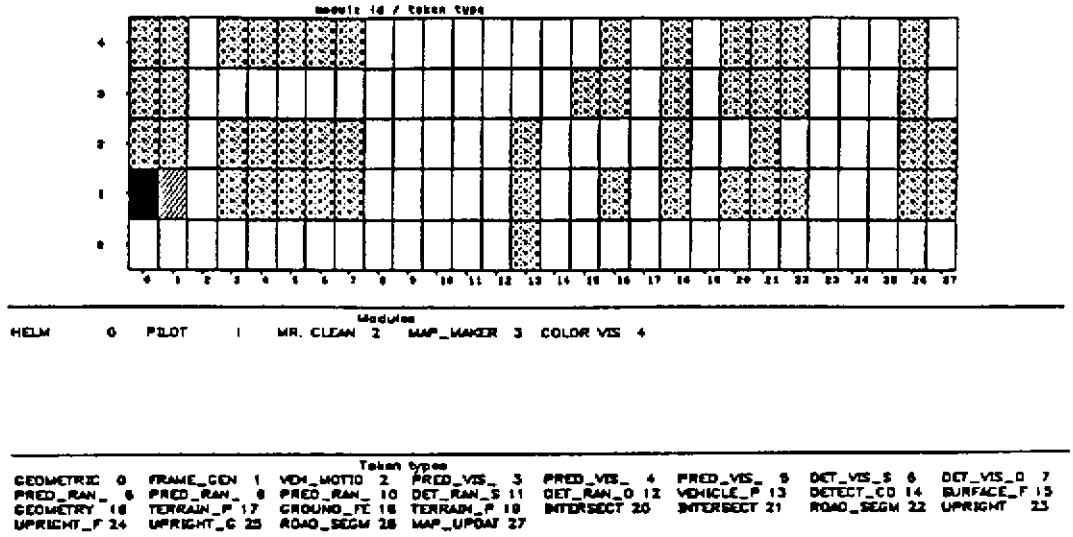


Figure 2: Interconnectivity of Typical CODGER Application

system grows and ideas are tested. A star-shaped configuration makes data re-routing easy without requiring new communication channels. Second, data created in one module is often used by many other modules. Figure 2 illustrates the "interconnectivity" of a typical CODGER application. In this example, the system contains five large modules that exchange data of 28 different types (tokens). A box is darkened in the grid if the module corresponding to the row reads and/or writes the data type corresponding to the column. The module and data type names are listed below the grid. Note that most of the data types are accessed by most of the modules in the system. The CODGER configuration precludes the need for a full clique of communication channels. Third, the CODGER communications scheme is very extensible in that it supports anonymity between modules. We can add a new module that taps into a communication line via the database (for example, a graphics module that displays data in the system), without making changes to the source or destination modules in the line. The main disadvantage to the star-shaped arrangement is that *all* data must pass through the local map. In some cases, a particular data type is designed to be passed between a pair of modules only. In such cases, the additional routing leads to slower execution.

CODGER is similar to a traditional blackboard system, in that it has a central database, a database manager, and a pool of modules that can read and write the database. Such systems are *heterarchical* because the "knowledge" in the system is distributed among a number of modules (often called *knowledge sources*), each of which can read or write any piece of the data in the database. Each knowledge source (KS) has a set of preconditions, or predicates defined over the database, which must be satisfied in order for the KS to execute. In a traditional blackboard, the database manager determines which preconditions are satisfied, selects one KS from the eligible set using a fixed priority scheme, invokes the KS as a subroutine, and repeats.

A traditional blackboard functions as a problem-solving framework. At any point in time, the database represents the current state of the solution. The knowledge sources use techniques such as *forward-* or *backward-chaining* to transform the database into a solution state. The database manager relies on a priority scheme to focus the activity of the KS's on those search paths likely to lead to a solution. This problem-solving framework is not needed for our navigation tasks, as the tasks are more algorithmic in nature. Instead, a system is needed that can effectively distribute the computation across multiple processors to maximize parallelism. The serialized control and execution scheme of a traditional blackboard is unsuited for a distributed computing environment. CODGER differs from these blackboards in a number of ways. The modules in the CODGER system run continuously and in parallel. Modules communicate by storing and retrieving data from the database. A module is able to run as soon as its precondition (request for data) is satisfied, regardless of the state of execution of the other modules. Synchronization is achieved by suspending execution within a module until the requested data appears in the database. CODGER even offers a mechanism whereby a module can perform one computation and be interrupted when a precondition is met to perform another. Thus, parallel processors can be effectively utilized. Preconditions in CODGER are not compiled prior to execution but can be changed by the modules dynamically during execution. This feature frees CODGER from a fixed control scheme, allowing it to change dynamically in response to different computing requirements. Furthermore, CODGER has no fixed priority scheme for activating modules. The LMB matches preconditions to the database using a FIFO scheme. Therefore, priority scheduling is left to the modules, to be encoded in the data itself. Overall, the system can be viewed as data-driven, where the data flow pattern can change during execution.

The ability of the system to change its control scheme and data flow dynamically is of special importance to the NAVLAB system. Certain modules in the system (particularly perception and planning modules) require different amounts of time depending on the NAVLAB's environment (e.g., road following, intersection navigation, off-road driving). By observing the behavior of these modules during execution, the Pilot acts as a "meta-module" and adjusts parameters in the data that flows between the other modules to maximize parallelism. This control scheme cannot be compiled into the system a priori.

Data Storage and Transfer

Since the NAVLAB development environment consists of multiple types of computers and programming languages, it is important to have a data format that can be understood and manipulated by all modules, regardless of the machine architecture or programming language. The fundamental unit of data in CODGER, the *token*, exhibits this property. As the system evolves, so does the data format used by the modules. CODGER isolates the impact of this restructuring, requiring only those modules directly affected by the change to be modified. In order to effectively utilize a multiprocessing environment, primitives must exist for routing the data and synchronizing the transfer between modules. CODGER employs pattern-matching for routing and a blocking/interrupting scheme for synchronizing transfer. Both the data format and transfer mechanisms are described in greater detail in this section.

Database Tokens

CODGER tokens consist of classical *attribute-value pairs*. Attribute names are strings and their values can be scalars (integer, float, boolean, enumerated types, and strings), arrays (including arrays of arrays), pointers (to other tokens), or geometric location (described in the next section). Tokens are the basic unit of data in CODGER and can be used to represent physical objects, predictions, commands, status

information, etc. Modules create, write, read, and delete tokens in the global database by calling functions in the LMB Interface. The LMB Interface takes care of transforming the data format between different hardware architectures and programming languages.

Each token *type* (set of attribute names and data types) is defined in a *template* file. At startup time, the LMB parses the template file to determine the format of all tokens in the system. The binding of attribute names (strings) to fields in the token occurs in each module when it connects to the LMB. Functions in the LMB Interface perform type checking during execution to ensure that data types are used correctly. Since operations are dynamic, the work needed to modify the system is minimized. Attributes can be added, deleted, or changed in a token type without requiring re-compilation of any modules except those that use the attribute. This feature is of particular importance since typical CODGER applications have consisted of over 100,000 lines of code.

The attributes themselves fall into three classes: internal, local, and global. Internal attributes are automatically included in every token type and specify information such as the token's type, unique ID number, creation time, most recent modification time, version number, and creator. These attributes are managed by CODGER but can be examined by the user modules. Local attributes are defined by the user modules in the template file. The scope of these attributes is restricted to the token type in which they are declared. Global attributes are also defined by the user but can be used in a number of token types. They are intended to have the same semantics (universal) regardless of the token type in which they are used. The real power of global attributes is that they permit modules which understand a given global attribute to manipulate token types that it has not been pre-programmed to handle. The module can accomplish this task without needing to "understand" the other local attributes of the token.

Synchronization Primitives

Modules can retrieve tokens from the central database by direct address (request by ID number) or through the use of a pattern called a *specification*. Specifications are boolean expressions defined across token attribute values. If a token in the database satisfies a specification, the token is sent to the querying module. For example:

if *surface* equals **navigable** and *boundary* equals **perceivable**

In this example, "*surface*" and "*boundary*" are global attribute names defined in the template file. The strings "**navigable**" and "**perceivable**" are scalar constants from an enumerated type, and "equals" and "and" are function names embedded in the specification language. This specification matches any tokens (possibly representing roads, intersections, or off-road patches) over which the robot can drive and the boundary of which it can perceive. One possible use of such a specification is for path planning. Note that above specification matches tokens of *any* type, provided they have the above global attributes with the appropriate values. In this way new token types can be added to the system without needing to modify the path planner. Routing via pattern matching is more powerful than a simple routing table because it channels data based on semantic interpretation, rather than data format type.

In addition to routing, the transfer of data between modules must be synchronized. Whenever a module sends a specification to the LMB for matching, it also selects one of the following synchronization modes:

- *Immediate Request:* As soon as the LMB receives such a request, it matches the specification against the database. The calling module blocks execution during this time. If one or more tokens matches, the tokens are sent to the calling module and it resumes execution. If there is no match, one of the following actions happens at the module's discretion:
  - *Non-Blocking:* The LMB responds that there are no matching tokens, and the module resumes execution.
  - *Blocking:* The module remains blocked until a token is deposited that matches the specification, at which point the token is sent to the module and it resumes execution.
- *Standing Request:* The calling module passes the name of a function (interrupt handler) along with the specification to the LMB Interface and resumes execution. Whenever an incoming token matches the specification, the LMB sends the token to the calling module. The LMB Interface in the module generates an interrupt and calls the interrupt handler passing the token as an argument.

An immediate request facilitates synchronous communication between modules. Such requests are typically used for modules expecting a certain type of data at a certain point in time or sequence. The non-blocking option allows the module to take action if "expected" data is not resident in the database (e.g., to prevent deadlocking). A standing request facilitates asynchronous communication. Such requests are typically used for modules which function as servers but cannot afford to block, such as a vehicle-control module which must monitor signals from the vehicle while fielding steering requests from other modules.

Because all of the modules have access to all of the data, CODGER employs a locking scheme to ensure that the database is consistent. Modules that wish to change tokens in the database must lock them upon retrieval. This mechanism ensures that two modules do not modify a token in parallel. If a module sends a specification to the LMB with a locking request, the matching tokens are not returned until they are unlocked. Thus, an immediate, blocking specification will block until a matching token is unlocked. Our approach is that the database should be consistent so that it appears to be "local memory" to the user module. Note that our locking scheme does not prevent deadlock; it is the responsibility of the modules to avoid or detect such a condition.

Geometric Representation and Reasoning

The ability to represent, manipulate, and index geometric data is important to any navigation system. Perception modules must be able to represent the shape and location of detected objects. Planning modules must be able to search a map to find the best path for the robot, and control modules must be able to field requests to sense data at a particular location and to steer the robot along a trajectory. Since nearly every module makes use of geometry, CODGER support must be distributed. However, in order to guarantee that the geometry used in each module is globally consistent, the support must include a centralized component. In this section, we describe CODGER's facility for representing and indexing geometric data and for ensuring global consistency.

Geometric Data and Indexing

Geometry in CODGER is a separate data *type*, called a *location*. A location consists of a basic shape and a coordinate frame in which the shape is expressed. The primitive shapes that CODGER supports are: points, edges, arcs, ribbons, and polygons. Although these shapes are of no higher dimension than two, by using the appropriate coordinate frames, a three-dimensional object (such an a polyhedron) can be constructed from a set of locations.

Because geometry is a data type in CODGER, it can be embedded in specifications for geometric indexing. The specification language supports a number of indexing functions, including Euclidian distance, polygonal intersection, inclusion test, and minimum bounding rectangle (MBR). Geometric indexing is used by planning and perception modules for operations like searching a map database to find the best sequence of road segments to a goal or for determining which map objects should be visible to the robot. A simple example of the latter operation is shown below:

if *objecttype* equals **mapobject** and (*tlocation*
poly-intersect **viewframe**) equals **true**

In this example, "*objecttype*" is an attribute which is set to "**mapobject**" if the object is part of the map database. The string "*tlocation*" is an internal attribute of type "*location*" which holds the shape and location of the map object. The string "**viewframe**" is a constant of type "*location*" which defines (using a polygon) the robot's field of view projected onto the ground. The function "poly-intersect" returns "**true**" if its two arguments intersect. Thus, the above specification returns all map object tokens that appear in "**viewframe**". It should be noted that the geometric indexing functions supported in CODGER are two-dimensional. Based on the navigation scenarios previously discussed, the ground around the robot can be assumed to be approximately planer. Thus, for local planning and perception operations, two-dimensional indexing is sufficient.

Frames and Frame Generators

As described in the previous section, geometric data consists of two parts: a shape and a coordinate *frame*. In CODGER, a frame consists of a base frame and a homogeneous transformation from the base frame to the frame itself. There are two system-defined coordinate frames: the WORLD frame affixed to the ground, and the VEHICLE frame affixed to the robot. User modules are able to define their own coordinate frames and to use them for expressing geometric data. The advantage of this feature is that a module can define coordinate frames that are convenient for expressing its data. For example, a perception module can define a coordinate frame affixed to the camera (relative to the VEHICLE) for representing detected objects. This is certainly more convenient than expressing detected objects in a world-based coordinate frame, especially since the robot is moving. CODGER provides a facility to allow the user to express any piece of data in any coordinate frame known to the system. This feature is useful for a planning module that needs to express perception data from multiple sources or times in a single coordinate frame for trajectory planning. When performing geometric indexing, CODGER automatically transforms all geometric data accessed into the same coordinate frame. Because the robot moves in its environment, the transformation between some frames varies over time. CODGER provides *frame*

generators for representing time-varying transformations. A frame generator is a function that takes a time parameter as input and returns a homogeneous transformation as output, to be interpreted as the transform between the frame generator's base and object frames at the specified time. CODGER supports one system-defined frame generator, namely the one between the WORLD and VEHICLE coordinate frames. In order to generate transformations between the robot and its environment, CODGER must be supplied with the sequence of arcs and vehicle speeds (called the *history list*) specifying the robot's trajectory relative to the world. Frame generators can be included in location data structures in place of a coordinate frame, provided a time parameter is provided somewhere in the chain of frames. For example, assume that an object is sighted by a perception module at time t. The module creates a location data object representing the object based on the camera frame at time t. The camera frame has been previously defined as fixed relative to the VEHICLE. When CODGER is called upon to determine the distance between the object and a landmark relative to the WORLD (possibly to determine if they are the same object), it uses the time parameter t to "select" a transformation between the VEHICLE and WORLD. Thus, CODGER is able to transform the object from the camera frame into the WORLD frame for the distance calculation.

Geometric Consistency and Affixment Groups

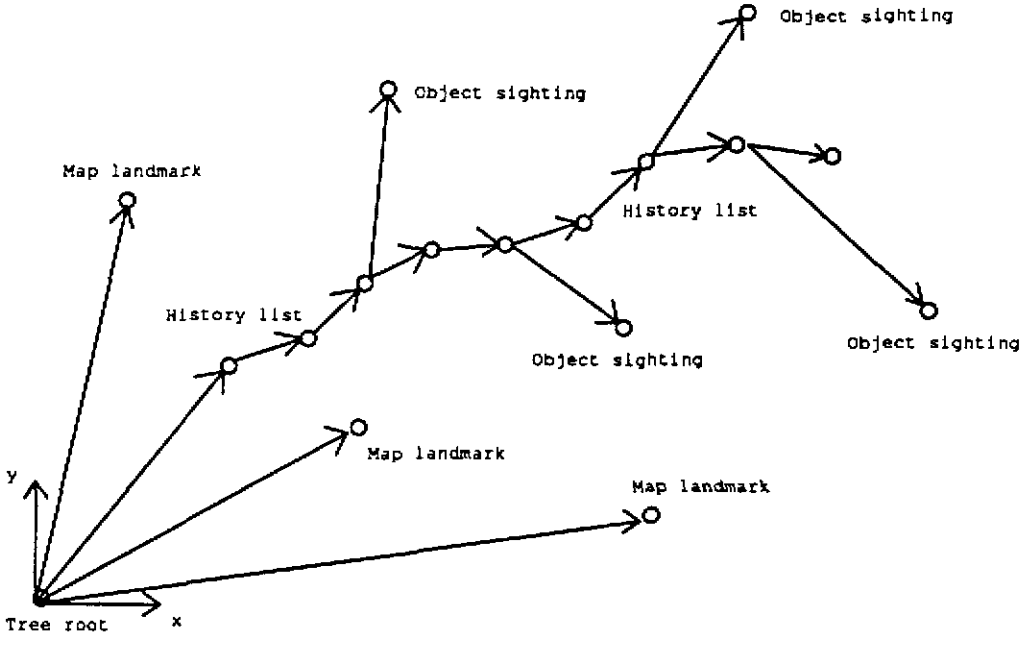


Figure 3: Tree of Coordinate Frames in CODGER

Provided that each frame in the system has only one base frame, the set of frames forms a tree, as illustrated in Figure 3. In this figure, the nodes are coordinate frames and the arcs are transformations between the frames. In this example, the central "spine" of arcs is the robot's history list. Arcs emanating from this spine are object sightings from the robot. The remaining arcs are landmarks in the map. Because of the tree structure, there exists a single and unique path between any two coordinate frames in the system. In some cases, it is desirable to have a coordinate frame with more than one base frame

and transformation. Such a case arises when an object is sighted multiple times or when a sighted object is identified as a landmark in the map. In such cases, the tree becomes a graph, and more than one path exists between pairs of frames. Unless all measurements are perfect, the paths will yield different transformations, and the graph is globally inconsistent. If we have a model for the discrepancy, the transformations can be adjusted to make the graph consistent [13, 48]. CODGER retains the tree structure (and thus global consistency) by discarding all but the most recent base frame for each coordinate frame. In general this strategy works well because the most recent measurements reflect the best estimate of a frame's transformation (e.g., the robot is closer to the object).

When an object's position is recorded relative to some frame at time t, a mechanism is needed for specifying how the transformation should be handled at times other than t. For example, assume an object is sighted from the robot at time t with transformation T. If we can determine that the object is moving relative to the WORLD, at time t' the transformation will be T'. However, if the object is not moving relative to the VEHICLE, the transformation at time t' will still be T. In order to disambiguate these cases, in addition to a time value the user can specify an *affixment object* when defining a coordinate frame. An affixment object is a coordinate frame such that the transformation to this frame from the new frame is constant over *all* time. Coordinate frames which are "tied" to the same affixment object are called an *affixment group*. If no affixment object is specified in a frame definition, then the transformation is assumed to be valid only at the given time.

Global Navigation

The task of global navigation assumes at least a minimum of information is known about the robot's environment a priori in order to plan a coarse path. The balance between that which is known a priori and that which is discovered as the robot moves determines the perceptual and planning strategies employed by the robot during navigation. At one extreme the robot may start with no map information. The robot's goal may be simply a distant point specified in a world-based coordinate frame. In this *map-building mode*, the robot *itself* searches for a path to the goal, using its sensors to discover the search space as it moves. A map can be constructed consisting of the perceptual data attached to the trajectory history of the robot for the various paths explored.

At the other extreme, the robot operates with a complete map of the environment. In this *map-navigation mode*, the robot is able to search its map database *before* navigating to determine the best route to the goal. While navigating the robot uses its sensors to repeatedly register its position with the pre-planned route. Of course, most realistic navigation scenarios fall in between the extremes. Navigation thus consists of a combination of exploration and registration as the robot moves.

Regardless of the mode employed, a map is needed for representing objects in the environment. The map developed for the NAVLAB is described in this section. We did not focus on research in global navigation algorithms for the NAVLAB. The algorithms we designed were guided by fairly general principles, although the particular implementations were ad hoc. In this section we discuss the principles employed and refer to related work in this area.

Map for Navigation

Whether the map is used as a database for navigation or a repository for sensor data, a representation scheme is needed for storing salient information for future extraction. There are two fundamental types of data any map for navigation needs to represent:

- *navigable geometry:* regions which are definitely navigable, definitely unnavigable, or possibly navigable. The navigable geometry in the map forms the core data from which a coarse, global path can be planned.
- *perceivable geometry:* regions which can be detected by one or more sensors onboard the robot. The perceivable geometry instructs the robot how to recognize landmarks and thus how to register its position relative to the map.

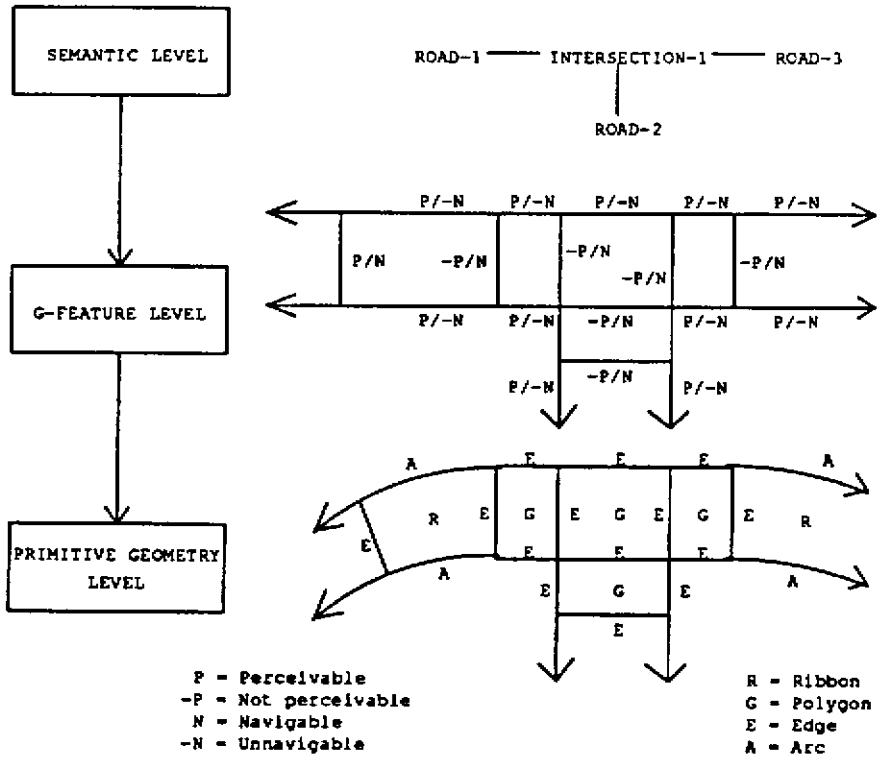


Figure 4: Three-tiered Map Representation Scheme

Using the above data types, we have designed a three-tiered map representation for on- and off-road navigation. The basic structure is illustrated in Figure 4. The top level represents objects with semantic significance to navigation. For example, objects such roads, intersections, trees, and landmarks are represented at this level. An intersection has semantic importance to navigation because it is a potential switching point from one road to the next. Objects at the semantic level are comprised of a number of *geometric features*, represented at the middle level. A geometric feature is an object subpart of uniform geometry that is tagged with a navigability label or perceivability label or both. For example, an

intersection is composed of a kernel polygon, describing the core of the intersection into which all of the connecting roads feed, and polygons defining a portion of the connecting road segments (see Figure 4). At the geometric feature level, all of these polygons are labelled as "navigable". The geometric feature polygons are further decomposed into geometric feature edges that bound the polygons. Those edges defining the connection between the kernel and the road segments are labelled as "navigable" and "unperceivable" while those defining the boundary between the intersection and the surrounding off-road regions are labelled as "unnavigable" and "perceivable" (assuming there is a perceivable material difference between the two). Each geometric feature at the middle level points to a single object in the bottom level. The bottom level specifies the exact geometry (if known) for the middle level. The basic geometric primitives at this level are "polygon", "ribbon", "edge", "arc", and "point". The semantic level can provide geometric information about an object that is redundant with the data at the bottom level; however, it is generally in the form of attributes such as "length", "width", "size", or "area" which summarize geometric information for purposes of planning a coarse path.

The main advantage of this representation scheme is that data of different modalities are factored out into separate levels. A global planning algorithm can examine data at the semantic level alone to plan a coarse path to the goal, without concerning itself with the specifics of exactly how the robot should drive through navigable subparts or position itself to see perceivable subparts. Likewise, a local planning algorithm can reason about exactly those issues the global planner does not, without concerning itself with what type of object it is driving across or attempting to perceive.

The token construct of CODGER was used to represent the map. Separate token types were used to represent the semantic objects, geometric features, and geometric objects at the three levels. Traits such as "navigability" and "perceivability" along with pointers defining the "subpart" hierarchy were represented using global attributes. The advantage of this scheme is that new types of semantic objects which exhibited these globally-understood properties could be added to the system without modifying many of the existing modules. For example, a path planning module could fetch a new type of object called "stairs", examine its "navigability", and decide whether or not to drive across it without ever needing to understand its other attributes.

Route Planning and Navigation

Route planning is the task of using the map to select a coarse path and monitoring its execution by a local navigator. In the event that the map is complete, the entire route can be planned before the robot begins moving. In this case, the robot uses its sensors to register its position relative to the route as it navigates. If the map is incomplete, the robot uses heuristics to attempt to determine the best route to the goal. As the robot navigates along the route, it uses its sensors to fill in the missing information. As the map becomes more complete, the robot may decide that another route is better than the current one, and backtrack to a previous decision point. Backtracking can result from extreme conditions such as a road completely obstructed by unmapped obstacles or by cost considerations, such as a road that takes an expected turn away from the goal. In the latter case, the robot decides that the expected cost of reaching the goal along the current road exceeds the cost of backtracking to and embarking upon another route. Note that with an incomplete map, the robot itself performs some of the searching for the goal. The primary difference between a robot search of the world and a computer search of the map database is that the robot incurs a cost for backtracking. This cost must be included in the search itself. Korf [33] illustrates how A* (heuristic search) can be modified to include the backtracking costs efficiently.

Because our navigation scenarios were not difficult on a global level, we developed a simple route

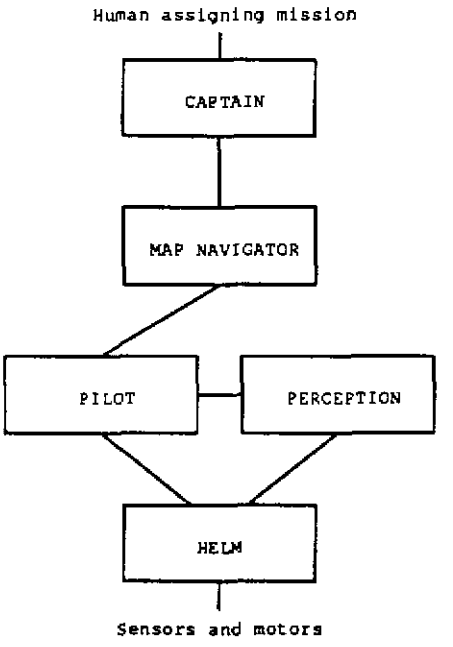


Figure 5: Module Structure for the NAVLAB System

planner and navigator that used a fairly-complete symbolic map to drive the NAVLAB and Terregator over a network of sidewalks and park roads. The map objects consisted of intersections, roads, and landmarks like trees. The map was complete topologically but not topographically, that is, the interconnectivity of the road network was specified completely, but the lengths of the various roads were not specified exactly (parameter intervals were used). Our global navigation system is described briefly here. The local navigation system, which was the focus of our research, is described in greater detail in later sections. Figure 5 illustrates the structure of our system. At the top level is the Captain, which receives a *mission* from the user. A mission is a sequence of goals specified by map symbols for the robot to visit. Goals are passed one by one to the Map Navigator, which searches the map database to find the best route to the goal. The Map Navigator decomposes a route into a number of *route segments*, or subroutes of uniform navigation. For example, the Map Navigator may find the following route to the goal:

1. Drive along Road 1 to Intersection 1.
2. Turn right at Intersection 1 onto Road 2.
3. Drive along Road 2 to Intersection 2.
4. Turn left at Intersection 2 and Stop.

Each road and intersection in this example forms a separate route segment, since navigation and sensing strategies differ on roads and intersections. The route segments are passed one by one to the Pilot, or local navigator. The Pilot examines the map to locate navigable and perceivable components of the route

segment and coordinates the use of Perception and the Helm to oversee execution of the route segment. Each module in the hierarchy reports success or failure to the module above it. A report of success notifies the parent module that the robot is ready to execute the next segment of the plan, while failure indicates that another route segment must be chosen.

The advantage of this system structure is that there is a clean separation between navigation at the global and local levels. Global navigation need only to be concerned with the selection of coarse paths to the goal, taking into consideration only semantic constraints (such as the requirement to drive on roads and stay off grass) and approximate geometric data. Local navigation, on the other hand, need only to be concerned about coordinating sensors and driving the robot to realize the plan.

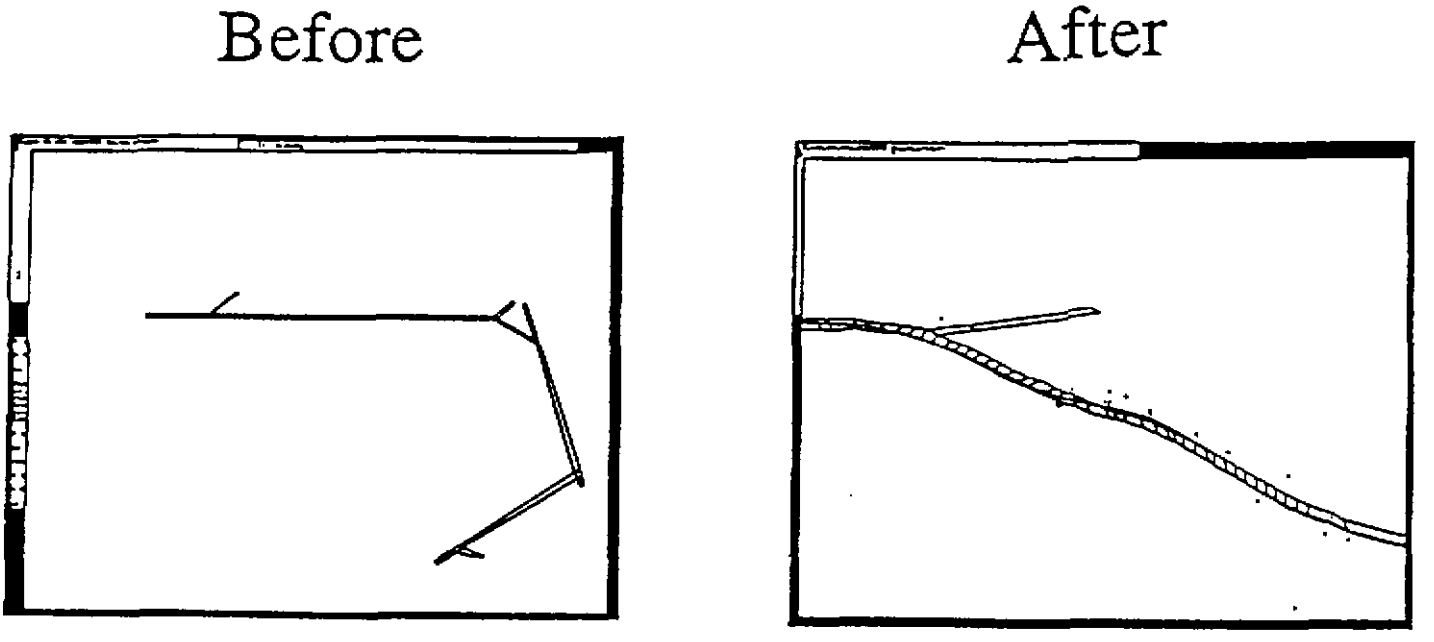


Figure 6: Map-Building for the NAVLAB on Park Roads

Data for a run on park roads is illustrated in Figure 6. The top diagram shows the topological map available to the system before the run. The bottom diagram shows the updated map, complete with topographical information. The trapezoidal polygons are the sensor viewframes for each camera image digitized, and the "dots" are trees detected by the range sensor.

Local Navigation

The division between the global and local navigation in the system is based on two principles. First, successful navigation depends heavily on features of the robot's environment that are too small or numerous to be stored in a global map. An obstacle on the edge of the road is one such example. Because they are not known a priori, they cannot come into play at the global planning level. They must be sensed and dealt with as needed as the robot drives. Second, characteristics of the robot such as kinematic constraints impact the robot's trajectory on a small scale (e.g., a few meters at a time), but have little overall effect on a coarse path planned at the global level.

Once a global path has been planned, it is the job of the Pilot to carry out each part of the plan or to report failure. To the global level, the Pilot appears as a black box. The input parameters consist of a recognizable goal, such as a landmark or a point in space relative to the starting location, and bounds on where the robot is allowed to move (route segment), based on semantic geometry such as a road. The output consists of either successful attainment of the goal or failure. To the Helm, the output of the Pilot is a series of robot trajectories and points for image digitization.

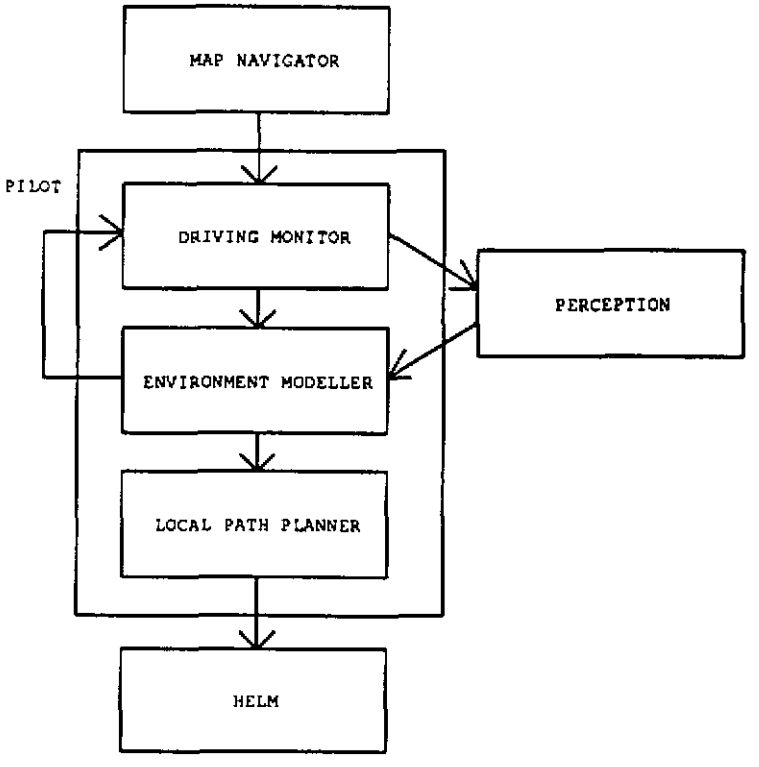


Figure 7: Local Operations on each Driving Unit

Before the robot can be moved forward safely, the environment must be sensed to determine its form if it is unknown or to determine the robot's position relative to it if the environment has been mapped a

priori. The area in front of the robot must be scanned for obstacles, and a trajectory must be planned to avoid the obstacles and to position the robot for the next sensing operation. These operations are carried out sequentially on a given patch of ground in front of the robot (called a *driving unit*) by the modules depicted in Figure 7. A route segment is traversed by driving across a series of driving units. For purposes of clarity, we assume that the robot is motionless while it senses and plans. Furthermore, we assume that driving units are of a fixed size and are non-overlapping. All of these assumptions are relaxed in the next two sections. The operations carried out at the local level are explained below via a description of each module.

The Driving Monitor

It is the job of the Driving Monitor to generate a prediction for the Perception module. The input to the module is the route segment within which the robot is to drive, a recognizable goal at the end of the route segment, and the position of the robot with respect to the route segment. The Driving Monitor retrieves the current description of the route segment from the map, examines the uncertainty in the geometry (polygon or ribbon) corresponding to the perceivable features (g-feature level in the map) to determine what portion will appear in the next image if any. All perceivable portions of the route segment that could appear in the next image are stored into a prediction token for Perception. Each object is assigned a number to be interpreted as the probability that the object will be found in the image. If the probability is high (near "1"), bounds are included on the positions of the edges to guide Perception.

Consider as an example the case of a road where the curvature is bounded between -0.01 and 0.01 meters^{-1} and the length is known to be between 40 and 60 meters. Assume that the road is terminated by an intersection. The Driving Monitor can use the bounds on the curvature to determine bounds on the appearance of the road edges in the next image. Likewise, if the robot has travelled 40 meters, the Driving Monitor predicts the appearance of the intersection, initially with a low probability that increases until the intersection is found.

The resultant prediction is sent to Perception. Perception uses the prediction to decide which recognition routines to invoke, and how to bound the search for these objects in the image. A request is also made to Perception to scan the driving unit for obstacles with the rangefinder.

The Environment Modeler

The Environment Modeler receives the results from Perception and updates the position of the vehicle relative to the map if there is no uncertainty in the map (complete-map mode) or updates the map objects such as the route segment if uncertainty exists (map-building mode). In the above example, if Perception reports that a road has been found in the image, the result can be used to remove the uncertainty in the curvature of the road for that driving unit. Thus, a better estimate of the shape of the road can be constructed incrementally by piecing driving units together. If the intersection is found, the road is terminated and the uncertainty in the length is eliminated.

The Local Path Planner

It is the goal of the local path planner to plan a trajectory to the next sensing point. The input to the planner is a planning space definition and a goal space. The planning space definition determines the area in which the robot is permitted to move in its attempt to reach the goal. Restrictions on the movement of the robot are of two types: semantic and kinematic. Semantic restrictions are determined by the navigable/unnavigable geometry of the route segment (at the semantic level in the map). For example, in road-following scenarios, we would like to restrict the robot to motion on the road, even

though the robot might be *able* to drive off the road. Kinematic restrictions are those pertaining to what the robot *can* do. For example, the robot cannot drive up stairs or over large objects. The planning space is defined by the union of a set of polygons, such that each polygon has one of the following labels: navigable, unnavigable, terrain-scan, and unknown. The labels "navigable" and "unnavigable" mean that due to semantic reasons (or kinematic reasons known a priori) the robot can or cannot respectively drive on the polygon. The label "terrain-scan" means that the polygon was scanned by a laser rangefinder and includes elevation data. The path planner is to evaluate the polygon for navigability as it plans based on kinematic considerations. Finally, the label "unknown" means that nothing is known (currently) about the navigability of the polygon. In the event that polygons overlap, the following precedence is enforced in the overlapping region:

$$unnavigable > terrain\text{-}scan > navigable > unknown$$

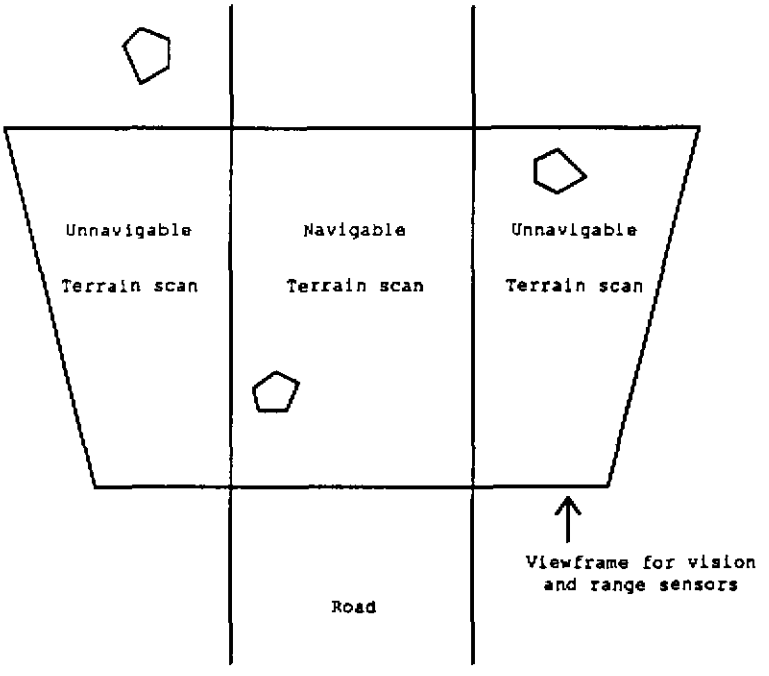


Figure 8: Precedence for Planning Space Boundaries

For example, consider a road-following scenario with obstacle avoidance. Figure 8 illustrates the "driving unit" after it has been recognized by the vision system and scanned by the laser rangefinder. From the vision processing, the road polygon is extracted and labelled as "navigable". The two polygons flanking the road are labelled as "unnavigable". The entire area scanned by the rangefinder is represented as a single polygon and is labelled as "terrain-scan". Note that there are three regions formed by intersecting polygons. The road area is labelled both "navigable" and "terrain-scan"; therefore,

"terrain-scan" has precedence. Thus, the local path planner is permitted to plan a trajectory anywhere on the road provided it is admissible kinematically (e.g., doesn't contact any obstacles or tip the robot), as determined from the terrain scan. The regions flanking the road also have two labels: "unnavigable" and "terrain-scan". The label "unnavigable" has precedence. Thus, the planner is not permitted to plan a path off-road even if kinematically possible from the terrain scan.

In addition to the planning space definition, a goal space must be selected before a trajectory can be planned by local path planner. The goal space is chosen to be the set of positions or *configurations* from which the robot is able to set the next image (road extension or landmark). By choosing a goal space instead of just a goal point, the local path planner has more freedom in selecting a trajectory.

Once the planning space has been defined and the goal space has been set, the local path planner plans a trajectory to the goal taking kinematic constraints into consideration. If the planner fails due to unsatisfiable kinematic constraints, the local path planner reports failure, and the global path must be altered or some other constraints (e.g., semantic) must be relaxed. The details of the local path planner are explored in greater detail in the next section.

The Helm

Once a trajectory has been planned by the local path planner, the trajectory is delivered to the Helm for execution. The Helm accelerates the robot from a stopped position to a preset speed (unless already moving at that speed), and then decelerates the robot near the end such that the robot stops at the end of the trajectory (unless a new trajectory is available).

The Driving Pipeline

All of the above operations must be performed on a given driving unit before the robot is permitted to drive across it. One strategy for coordinating the operations is for the robot to remain motionless while each operation is performed sequentially. Only when all operations have finished is the robot permitted to drive. Such stop-and-go sequencing formed the basic control strategy for a number of mobile robots [21, 40, 43, 50]. The primary disadvantage of such a strategy is overall system speed. If T_{tot} is the total time needed to perform all of the local operations (including driving the robot), and D is the length of the driving unit (the distance from the robot to the far boundary of the sensor's field of view), then the net velocity of the robot is $V = \frac{D}{T_{tot}}$. The inefficiency of this strategy is especially apparent in a multiprocessing environment. Since the operations are performed serially, there is no benefit to having more than one processor. Furthermore, when the robot is actually moving, no processors are in use. This execution pattern is illustrated in Figure 9. In this figure, the "bars" indicate the time during which a local operation is executing for the driving unit numbered above it.

We have devised a control strategy, known as the *driving pipeline*, for parallelizing the computation for implementation on a multiprocessor, thus increasing the speed of the system. Although local operations on any given driving unit must be performed serially, at any given point in time we can perform operations on *different* driving units in parallel. The pipeline is configured by running each operation on a different module, interconnected via the CODGER database. As each driving unit is created, it is deposited in the blackboard with a parameter set to indicate which operation should be performed on it next. Each local operation module matches driving units with the parameter set to its identification value. After the module

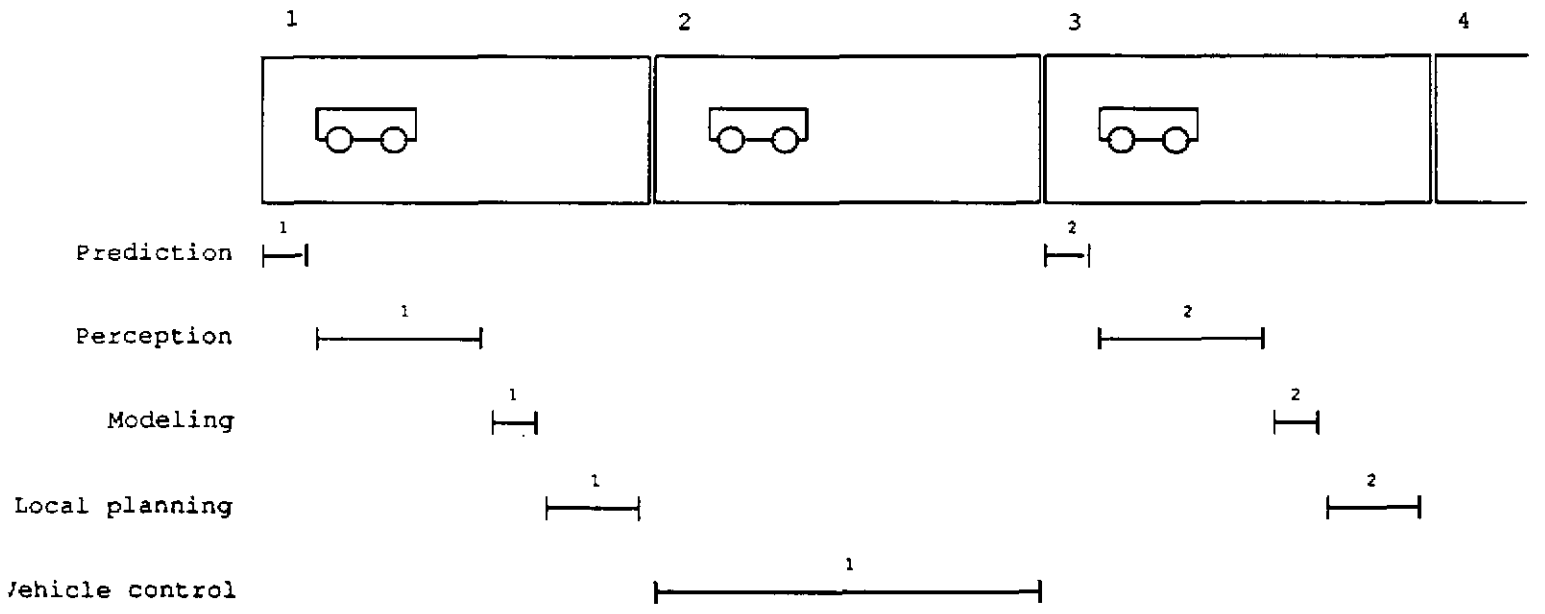


Figure 9: Execution Pattern in Stop-and-Go Mode

processes the driving unit, it advances the parameter and redeposits it. In this fashion driving units move sequentially from one operation to the next via the database. This process is illustrated in Figure 10. Note that at any point in time, the system processes the driving units in parallel. Since one of the operations in the pipeline is the actual driving of the robot, under ideal conditions (defined below), the robot moves continuously. Unlike other systems with parallelized control [39, 51], there are no rigid time constraints under which a stage (operation) of the pipeline must finish processing. If a stage bogs down on a driving unit, the succeeding stages stop and wait for the driving unit to "appear" in the database. In extreme cases, the robot will actually come to a stop and wait for the processing to finish. This feature is important for two reasons. First, different environments require different amounts of processing. For example, the perception module requires more time to recognize an intersection than a road, and the planning module requires more time to find a trajectory through a cluttered environment than through a flat, open road. Second, general purpose computers can be used to implement all of the local operations, since no real-time constraints are imposed on the modules.

Maximizing Parallelism

In theory, the use of the driving pipeline can increase the net velocity of the robot by a factor of $\frac{T_{tot}}{T_s}$, where T_s is the stage time of the pipeline (time of the longest stage). This increase assumes, of course, that enough processors are available to run all of the stages in parallel. Figure 11 illustrates pipelined execution. Note that at any point in time the stages process different driving units in parallel.

In practice, however, there are tradeoffs involved in maximizing parallelism. For example, in order to

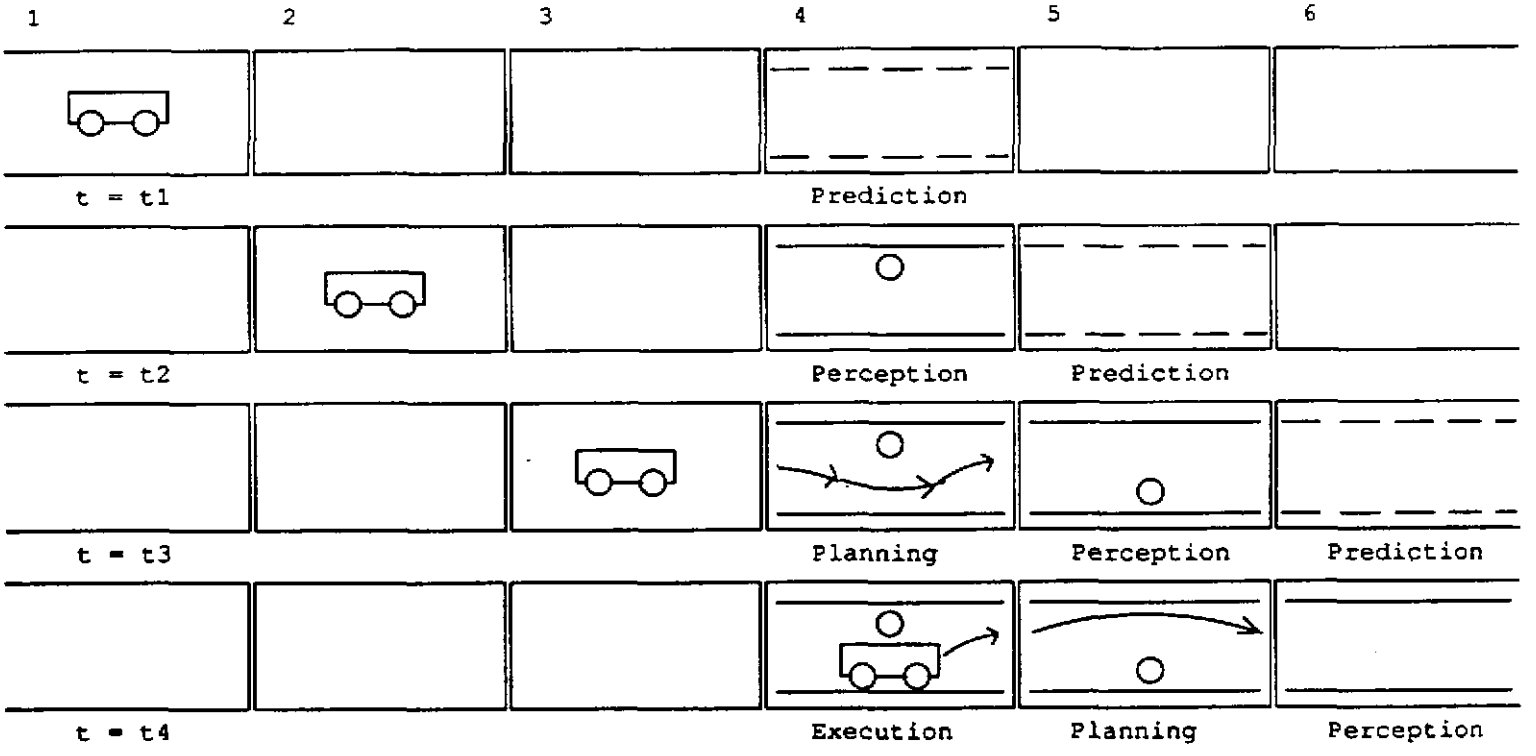


Figure 10: Pipelining of Local Operations

process multiple driving units in parallel, the robot's sensor must be aimed farther forward than in the stop-and-go case. This adjustment may render the sensing task more difficult or unreliable. In this section, we identify the parameters necessary for maximizing parallelism and explore the tradeoffs involved. We begin by relaxing the constraints that the driving units be of the same size and non-overlapping. Define the following parameters: Let T_m, T_p, T_e, T_l, and T_h be the average times required by the Driving Monitor, Perception, Environment Modeler, Local Path Planner, and Helm respectively to process one driving unit. Let T_{tot} and T_s be the sum and maximum of these times respectively. Let D_p be the driving unit length for Perception, S_p the offset distance from the robot to the proximal boundary of Perception's driving unit, and let D_i be the interval distance, that is, the distance the robot drives per cycle. The maximum velocity of the robot, V, is given by:

$$V = \frac{D_i}{T_s} \tag{1}$$

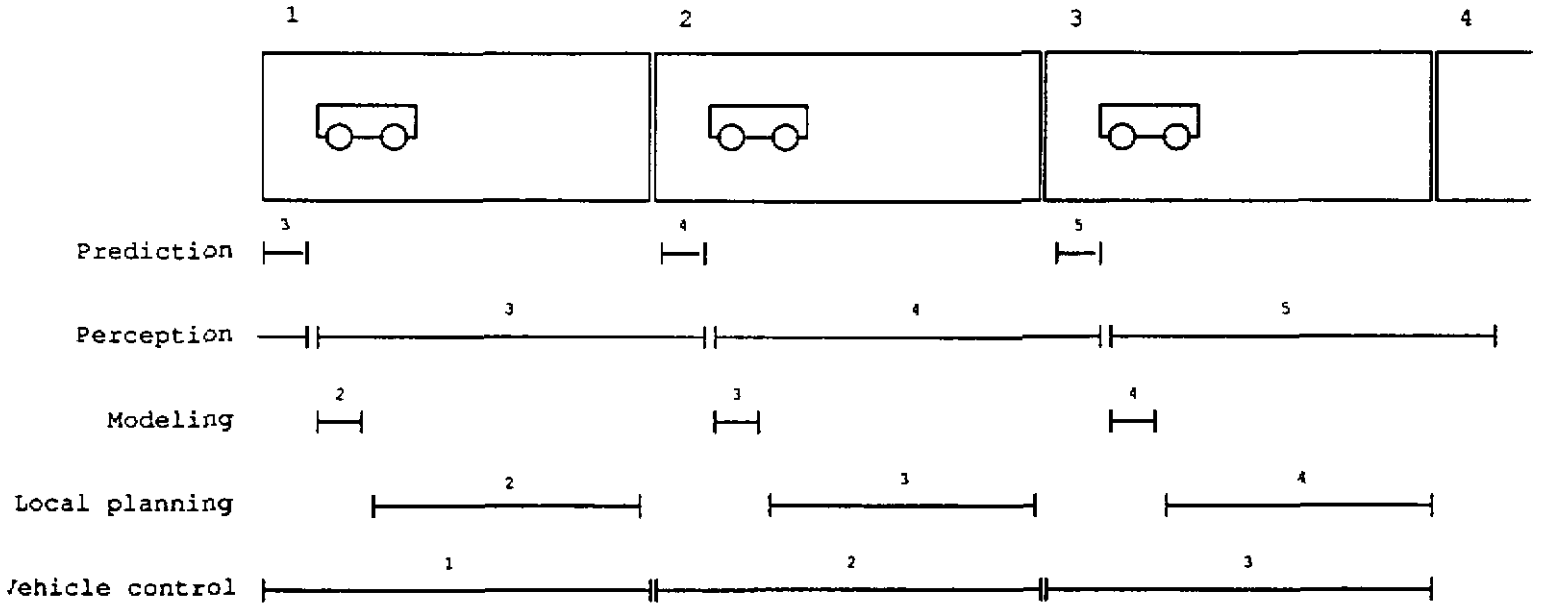


Figure 11: Timing Diagram for Ideal Pipelined Execution

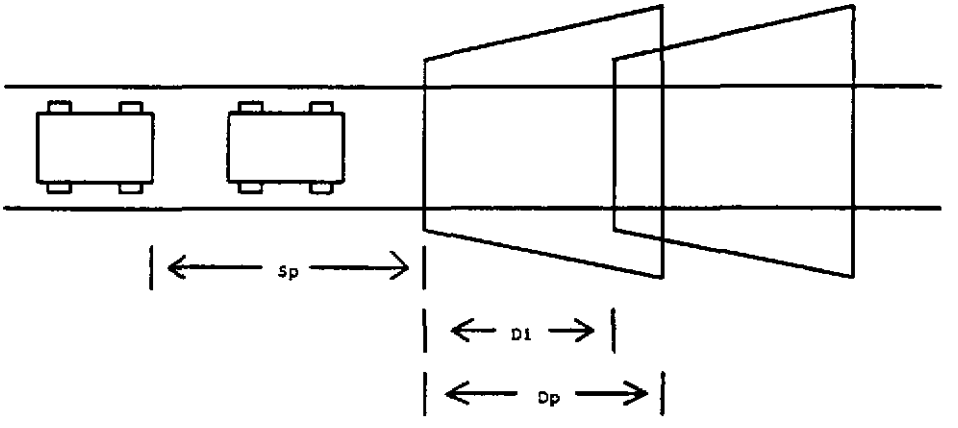


Figure 12: Pipeline Parameters

These parameters are illustrated in Figure 12. In order to ensure that the robot does not move onto a patch of ground that has not cleared all of the local operations, the interval distance, D_i, must be less than

or equal to D_p; otherwise, "gaps" appear in the spatial sequence. It is assumed that the processing time, T, for each module is constant regardless of the size of the driving unit, D, that it processes. This assumption is valid if the resolution of the operation is scaled to the size of the driving unit. The remainder of this section describes how the adjustment of the above parameters affects parallelism of the computation and performance of the robot on a per module basis.

Perception

Since Perception is one of the first operations performed on a driving unit, the offset distance, S_p, must be relatively large. Let V be the velocity of the robot and T_{sub} be the sum of the stage times from Perception through Local Path Planning ($T_{sub} = T_p + T_e + T_l$). In order to maximize parallelism in processing and minimize changes in robot velocity, all local operations on Perception's driving unit should finish immediately before the robot drives onto it, that is, $S_p = VT_{sub}$. Substituting the maximum velocity of the robot (Equation 1) into this equation, we have:

$$S_p = \frac{D_i}{T_s} T_{sub} \tag{2}$$

If S_p is set to be less than the above equation, parallelism is maximized but the robot degenerates to stop-and-go motion. If S_p is set to be greater, the robot moves continuously, but the processors are inefficiently utilized. Figure 11 illustrates the case where S_p is set according to Equation 2.

Note that the selection of the parameter D_i affects the Perception module. The greater the value of D_i, the greater the value of S_p and D_p (since D_p must be greater than or equal to D_i). Thus, in order to facilitate a higher vehicle speed, the robot's sensor must be aimed farther in front of the robot and must cover a larger area. At longer range, fewer pixels define the driving unit, thus reducing both the accuracy and reliability of the sensor reading. Furthermore, in situations requiring much maneuvering, such as turning corners, the S_p must be reduced to allow the robot to see the area at close range. In such cases, the robot's speed must be reduced correspondingly.

The Driving Monitor and Environment Modeler

Although the Driving Monitor and the Environment Modeler are separate modules in the system, there are dependencies in their functions. The Driving Monitor creates predictions for Perception based on the map. The Environment Modeler changes either the map itself (in map-building mode) or the robot's position relative to the map (in map-navigation mode). In the pipeline configuration illustrated in Figure 11, the Driving Monitor creates predictions in the i-th cycle from the map updated in the $(i-1)$-th cycle. In map-navigation mode, this discrepancy means that the positions of predicted objects lag one update behind the current cycle. In map-building mode, the discrepancy is more severe. Objects seen in the i-th cycle will not be predicted again in the $(i+1)$-st cycle. If the effects of this discrepancy are intolerable (as was the case on sharp turns where the straight-line approximation breaks down), the execution of the Driving Monitor on the $(i+1)$-st driving unit can be delayed until the Environment Modeler completes execution of the i-th driving unit. This synchronization pattern is illustrated in Figure 13. Note that since the Driving Monitor, Perception, and the Environment Modeler are serialized, they effectively become a single stage in the pipe. In this arrangement, if the sum of their processing times $T_m + T_p + T_e$ exceeds

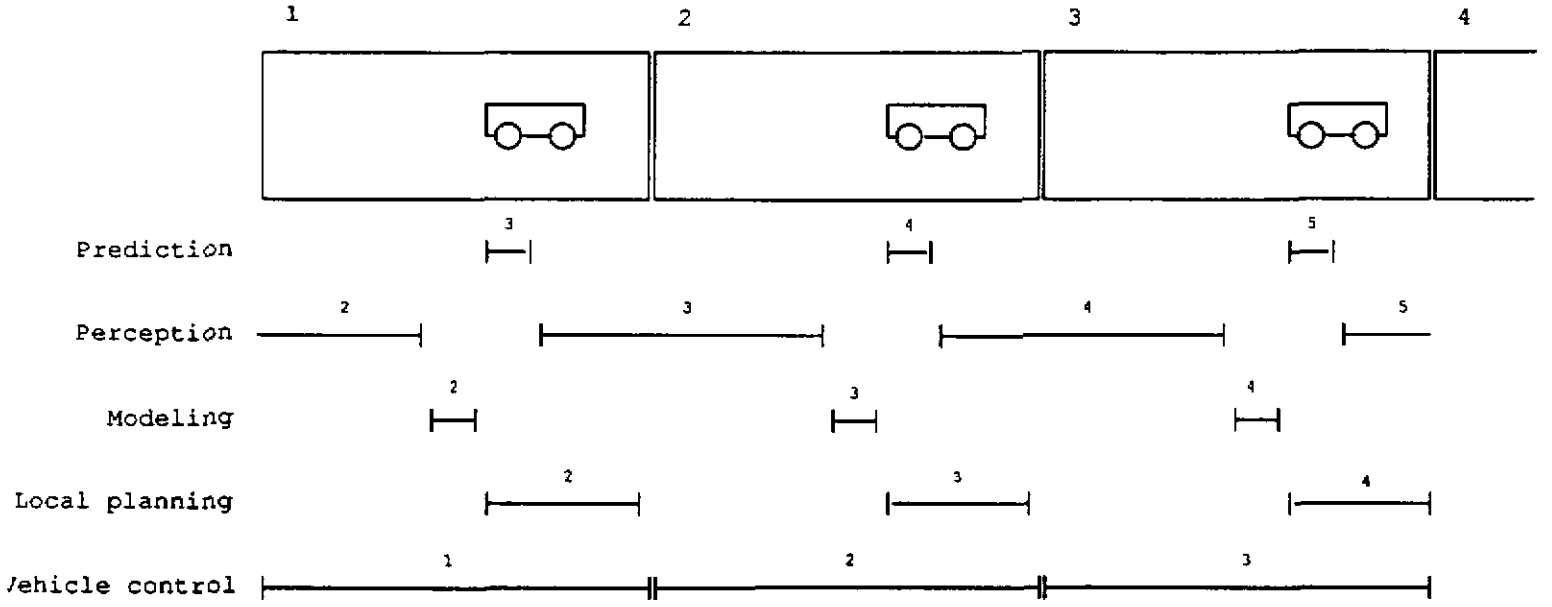


Figure 13: Synchronization of Driving Monitor and Environment Modeler

the current stage time, T_s, of the pipeline, the offset distance for Perception, S_p, will increase (see Equation 2).

The Local Path Planner

In the Local Path Planner, pipelined execution leads to less responsiveness in the robot. In the stop-and-go case, the Local Path Planner plans a path from the robot's current position to the distal end of Perception's field of view. The size of the planning space, D_l, is $S_p + D_p$. Although only the first D_i of the path is actually executed, the remainder of the path influences this first section, particularly in the presence of obstacles. In the pipelined case, the robot continues to move after digitizing an image as Perception, the Environment Modeler, and the Local Path Planner process the new driving unit. Thus, the size of the planning space is reduced by the distance travelled:

$$D_l = S_p + D_p - VT_{sub} \tag{3}$$

If the maximum velocity is chosen, $V = \frac{S_p}{T_{sub}}$, then the above equation reduces to $D_l = D_p$. For a given choice of S_p and D_p, the size of the planning space can be increased by choosing a V less than the maximum. The extremes are shown in Figure 14. In this example, for clarity S_p is chosen to be approximately two viewframes in length ($2D_p$), the average times for Perception and Local Path Planning

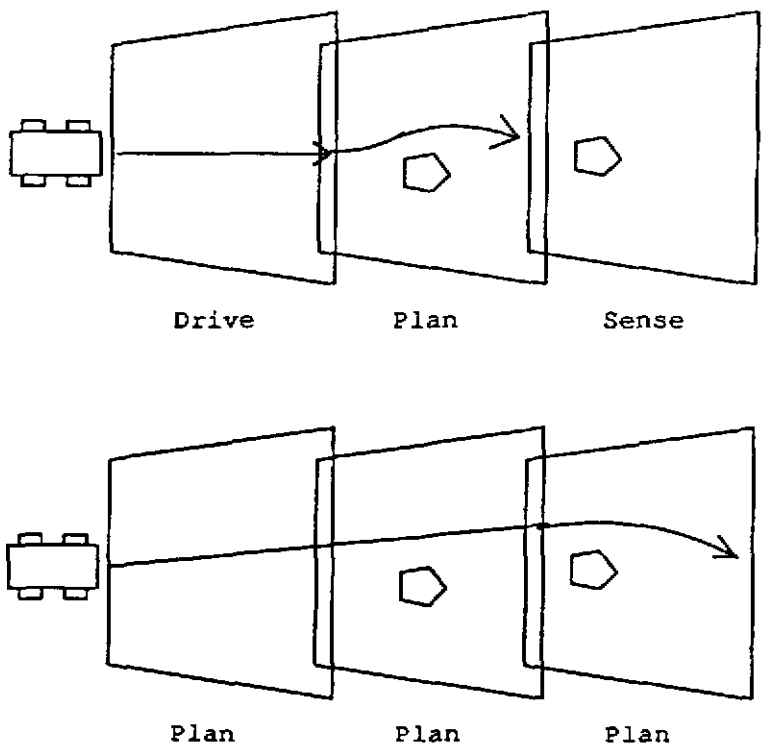


Figure 14: The Effects of Pipelining on Robot Responsiveness

are equal ($T_p = T_i$), and the other processing times are near zero. In the first configuration, the velocity V is set to the maximum. In parallel, the robot drives through the first driving unit, plans through the second, and processes sensor data in the third. The size of the planning space is one driving unit. In this example, the robot plans around the first obstacle, but unwittingly positions itself in front of the second. On the next local cycle, it will be unable to plan around the second obstacle and will be forced to back up. Thus, the robot is not very responsive to its environment. In the second configuration, the velocity V is set to zero (stop-and-go mode). While motionless, the robot senses the third driving unit and plans through all three, then drives through the first. Because the planning space is three driving units in length, the robot has enough room to maneuver around both obstacles.

The size of the planning space needed depends on the expected environment. Let R_{min} be the minimum turning radius of the robot, and W be the width of the robot's body. From Figure 15, the maximum size (width) of an obstacle that the robot can avoid without backtracking is given by:

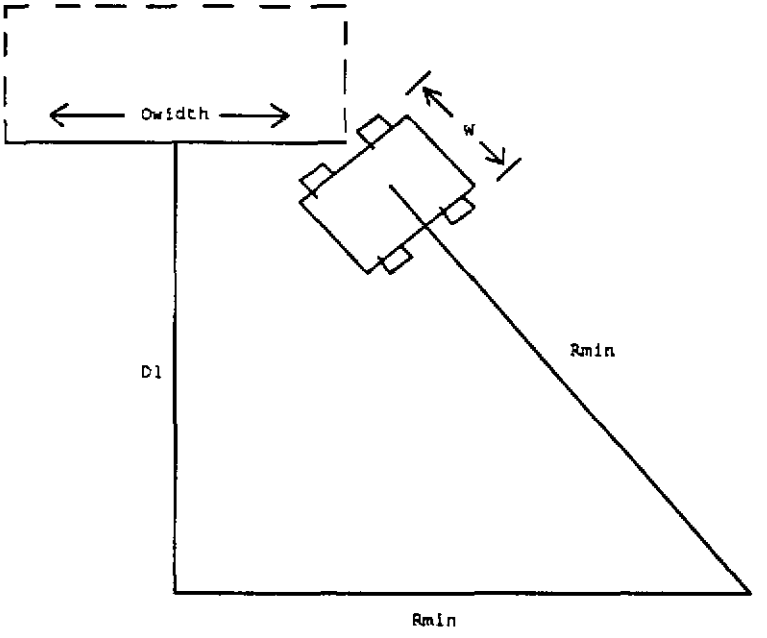


Figure 15: Maximum Obstacle the Robot can Negotiate Without Backtracking

$$O_{width} = 2(R_{min} - \sqrt{(R_{min} + \frac{W}{2})^2 - D_l^2}) \quad (4)$$

if $D_l \leq R_{min} + \frac{W}{2}$ and $O_{width} = \infty$ otherwise. In addition to reducing the maximum-sized obstacle around which the robot can maneuver without backtracking, a smaller D_l also leads to slightly longer trajectories, since the robot cannot "see" far enough in front of it to take straight-line trajectories around obstacles. Of course, this loss could be recovered by moving Perception's field of view farther in front of the robot, but only at the expense of the tradeoffs previously discussed.

The Helm

The advantage of pipelining within the Helm is that if the parameters are set properly, the robot will move continuously. As given in Equation 1, the ideal velocity for the robot is the driving interval divided by the stage time of the pipe. Unfortunately, this equation assumes that the stage time of the pipe never exceeds T_s. If it does, the robot must stop instantaneously to avoid "overdriving" the current trajectory. Since this requirement cannot be met with a real robot, the Helm drives the robot at a velocity V' less than V, such that robot has room to decelerate to a stop in the event that the stage time exceeds T_s.

Let D_1 be the distance the robot travels at the constant velocity, V', for T_s time:

$$D_1 = V' T_s \tag{5}$$

If the pipeline fails to deliver the next driving unit to the Helm at this point in time, the Helm decelerates the robot to a stop within a distance D_2. Since the robot must never overdrive its driving unit, we have:

$$D_h = D_i = D_1 + D_2 \tag{6}$$

Let A be the maximum deceleration constant of the robot. Since the robot must decelerate from velocity V' to zero within a distance of D_2, we have:

$$D_2 = \frac{V'^2}{2A} \tag{7}$$

Substituting Equations 5 and 7 into 6, solving for V' and discarding the meaningless root gives us:

$$V' = -A T_s + A\sqrt{T_s^2 + \frac{2D_i}{A}} \tag{8}$$

From the above equation, the difference between V and V' is determined primarily by A. If the magnitude of A is large, then the robot can stop quickly and V' is approximately equal to V. The net effect of running the robot at velocity V' rather than V is that on average the pipeline will be idle while the robot traverses the distance D_2, so parallelism is traded off for a guarantee that the robot will always stop in time.

Figure 16 illustrates execution data for a real run. In this run, the Driving Monitor predicted roads and/or intersections for Perception, which used a color camera to match the templates. The Local Path Planner positioned the robot for the next camera picture while keeping the robot on the road, and the Environment Modeler registered the position of the vehicle with a topographical map. In this configuration, the Environment Modeler lagged behind the Driving Monitor and Perception. While this

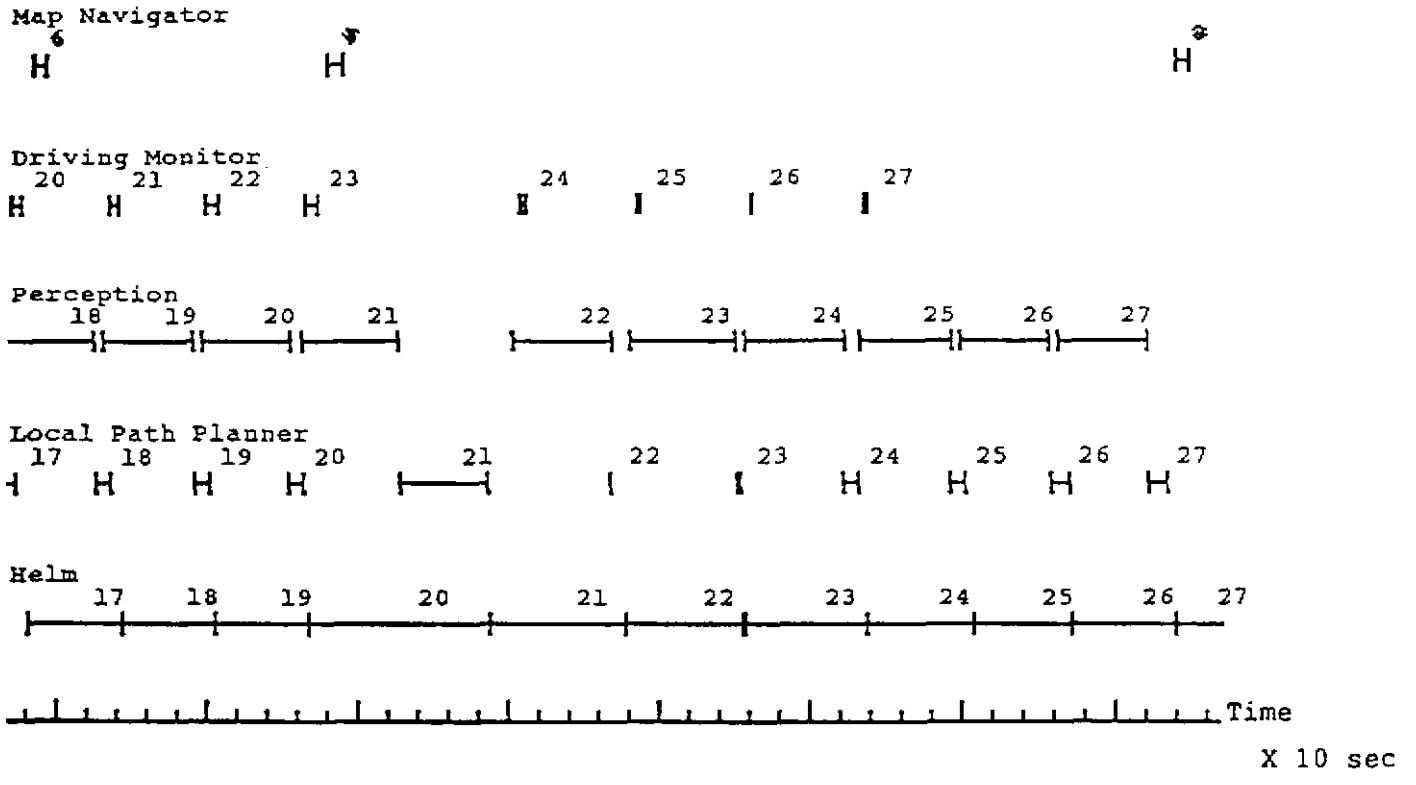


Figure 16: Real Pipeline Execution for Park Run

configuration increased parallelism and thus the robot's speed, the system was less stable since predictions for Perception were less accurate.

Section III
Kinematic Path Planning for Wheeled Vehicles

Introduction

Representing the Local Environment

The local environment of a mobile robot is that portion of the world in which the robot acts between sensing operations. Within each cycle of the driving pipeline, the Local Path Planner determines bounds on this environment based on semantic information such as edges of roads and intersections and periphery information such as the bounds on the sensor viewframes. The local environment is a combination of coarse, map-based information and information acquired from sensors that is too dense to be represented in a map. The robot must represent the environment and its relationship to the environment in such a way that it can determine safe and permissible areas in which to move toward its goal. A number of issues come to bear on the problem:

- A large number of parameters are needed to represent the robot's size and shape, its relationship to its environment, and the environment itself. Attempting to work with the complete set renders the planning task intractable. Instead, a smaller set must be selected that is manageable enough to plan with yet still guarantees safe passage of the robot.
- A mobile robot is generally not permitted to drive everywhere within its local environment. Certain positions will bring it into contact with physical objects, such as trees. Other positions will render the robot inoperable, such as on a sloped surface that would cause the robot to tip. Such positions are considered inadmissible. The robot must be able to identify these inadmissible positions in order to avoid them.
- In order for a robot to safely move through its environment, it must pass through a continuum of positions. Testing positions point by point is prohibitively expensive. Instead, large areas of the environment must be tested for admissibility in a single computation.

Existing systems address one or more of the above issues but make simplifying assumptions about the remainder. The robot's local environment is often simplified by modeling it as a field of polygonal obstacles arranged on a flat surface. The robot's shape is often modeled as a circle. We have developed a representation scheme for the robot and its environment that addresses all of these issues. In this section we describe this scheme. We begin by describing the space for representing state information about the robot and its environment. From there we define admissibility and how to compute it. Finally, we describe efficient techniques for calculating the admissibility for a set of robot positions in a single operation. There is, however, an issue that we do not address. We have assumed a static world throughout, i.e. the planner assumes that obstacles in the environment cannot move. As described in an earlier section, this assumption enables us to remove all real-time considerations from the system.

Local Path Planning

Global planning is the task of generating a coarse path based on a priori map information. Typically, an exact path cannot be planned because the map information is not complete. Sensors must be used to fill in the missing information as the robot moves through the environment. Local planning is the task of generating a trajectory for the robot through the sensed environment without violating the bounds of the global plan. Since sensor information is acquired incrementally, the global plan is realized through a series of such "locally planned" trajectories.

A number of issues come to bear on the problem at the local level:

- Robots are typically not omnidirectional. For example, a car-like robot cannot translate sideways. The robot must plan trajectories that it can execute, that is, it must take kinematic constraints into consideration.
- It is desirable for a mobile robot to navigate smoothly. Planned trajectories should not needlessly subject the robot to excessive wear and tear. Furthermore, jerky trajectories may cause instability and should be avoided.
- In addition to planning a trajectory for the purpose of travelling to some destination, the mobile robot may need to position itself in such a way as to keep a particular landmark visible, or to guarantee that all ground area to be traversed has been scanned by the sensors, that is, other geometric constraints may come to bear on the trajectory of the robot.

Existing systems address one or more of the above issues but make simplifying assumptions about the remainder. Typically, the robot is assumed to be omnidirectional. Smoothing is usually not addressed; neither are other geometric constraints such as positioning for sensing.

We have developed a planning algorithm that addresses all of these issues. In this section we describe the algorithm. We begin by describing a technique that handles kinematic techniques by fitting trajectories into bounded, admissible space. We then show how this technique can be combined with a standard search algorithm to plan paths. Planning for sensing is considered next, followed by path smoothing. Finally, there is an issue that our planner does not address. Since the robot's environment is assumed to be static, it can move as slowly as need be, so that dynamic constraints such as maximum velocity and acceleration can be ignored.

The Planning Space

The Configuration Space

Generally a large number of parameters are needed to describe a robot, its environment, and the robot's relationship to its environment. For example, parameters are needed to specify the physical dimensions of objects such as trees and rocks in the environment. Others are needed to specify the size and shape of the robot itself, and to specify the position and orientation of the robot relative to its environment as a function of time.

The parameters can be divided into two classes: those that cannot be controlled and those that can. Examples of the former include the physical parameters specifying the robot's shape, assuming we do not intend to plan beyond a crash, and those parameters specifying large, immovable objects, such as trees. An example of the latter is the position of the center of mass of the robot relative to the world. For those parameters that can be controlled, we can select a subset of independent parameters from which the remaining dependent parameters can be computed. For example, the position of a camera on the robot relative to the world changes as the robot is commanded to drive forward. However, the new values can be computed from the position and orientation of the center of the robot along with the fixed, size and shape parameters of the robot.

The entire state of the robot and its environment can be represented completely by this independent set of controllable parameters or *degrees of freedom*. A particular instantiation of these parameters (an N-tuple) is a *configuration*, and the collection of all possible configurations is the *configuration space*

[36, 38].

Factoring Out Parameters

The set of parameters needed to describe the robot and its environment can be quite large. Attempting to model all of these parameters in the planning process could render the task intractable. To reduce the complexity, we can select a subset of parameters, but we do so at a cost.

Assume we need N parameters to completely specify the robot and its environment. Assume we have a predicate P that returns *true* if a given N-tuple, or configuration, is safe and *false* otherwise. If we select a subset M, then there are $Q = N - M$ parameters we are not modeling. There are two types of mistakes we can make: First, we can assume a value of true for P for an N-tuple that is false. Second, we can assume a value of false for an N-tuple that is true. The first mistake is far more serious. Misclassifying an unsafe configuration as safe could jeopardize the robot. The second mistake could cause the robot to avoid areas that are actually safe. Since it reduces the amount of space the robot perceives as safe, the robot's ability to maneuver could be hindered. Since we are not modeling the entire N-tuple, we cannot eliminate both types of mistakes, but we can eliminate one at the expense of guaranteeing the other. We choose to eliminate the first mistake, thereby causing the robot to err on the conservative side, i.e. ensuring safety at the expense of some maneuverability. In order to avoid returning true for an N-tuple when P is false, we return true only when it is *impossible* for P to be false. For a given M-tuple, we return true only if P returns true for *all* configurations defined by the M-tuple crossed with the space of possible Q-tuples.

Consider the following example. For the purpose of collision detection, mobile robots which navigate on flat surfaces are often modeled by vertical cylinders which bound the shape entirely. The test for collision reports the same result at a given position (x,y) on the surface regardless of the orientation of the robot, θ. The parameter θ has been factored out of the planning problem. Note that the test errs on the conservative side, that is, it always reports a collision when one would occur; however, it also reports a collision for some values of θ that are clear.

The NAVLAB Configuration Space

The environment in which the NAVLAB navigates is modeled as unstructured three-dimensional terrain. For each point (x,y) in the local environment, the robot is able to sense the maximum elevation, or z-value. Generally, the world is assumed to be rigid, that is, a z-value will not change (decrease) under the weight of the robot.

The NAVLAB is approximately box-shaped. For the purpose of collision detection, we need only test the intersection of the terrain (z-values) with the robot's shape. Since the terrain model stores only *maximum* elevation values, using a *lower* bound on the height (z) of the robot at any point (x,y) guarantees that the collision test never reports "clear" when there is actually a collision. The shape is approximated by projecting all faces into the plane of the bottom, or undercarriage, of the robot (see Figure 17). This permits us to factor out all of the parameters describing the robot's exact shape, and replace them with a small set of parameters specifying a polygon in three-space.

For a robot such as the NAVLAB, there are a number of parameters that specify its relationship to the world. The NAVLAB is a rigid, unarticulated robot; therefore, its configuration in three-space can be

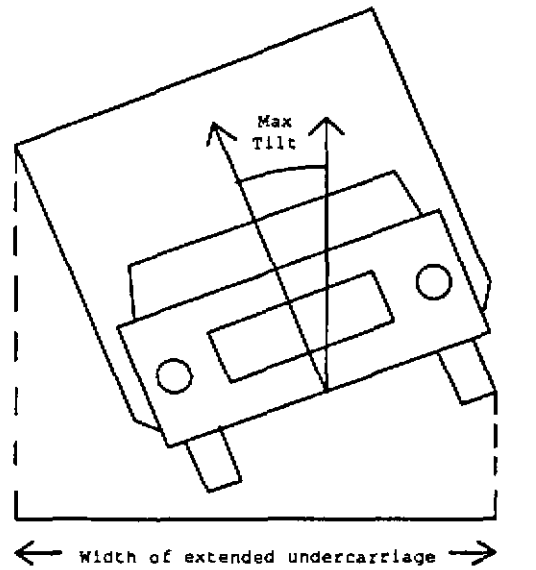


Figure 17: The Shape Approximation for the NAVLAB

represented by six parameters, three in position (x, y, z) and three in orientation (α, β, θ). Since the robot is mobile, it has velocity and acceleration parameters as well. The steering angle parameter determines the arc along which the robot is currently travelling.

Not all of these parameters are independent. Since the NAVLAB is constrained by gravity to rest on the terrain, the composition of the terrain with two translational parameters (x, y) and the heading angle θ completely determines the roll and pitch angles (α,β) as well as the elevation z of the robot. We assume that the maximum acceleration and velocity of the robot are small enough that they do not restrict the class of trajectories the robot can execute or the types of terrain over which the robot can drive. For example, we need not worry about tight turns that would result in the robot tipping at a high velocity. Therefore, these dynamic parameters can be factored out of the configuration space. Since the steering angle can be set to any admissible value at any configuration in the space, it also can be factored out. The only remaining independent parameters are (x, y, θ). These parameters comprise the robot's configuration space.

Space Admissibility

Admissibility Defined

Given that the robot has some configuration space in which to plan, planning is the task of selecting a path through this space from a starting configuration to a goal configuration. In order to accomplish this, the robot must move through a continuum of configurations (M-tuples). Generally, not all configurations are safe. A configuration is considered to be *inadmissible* if either of the following conditions holds:

1. It incapacitates the robot, rendering it unable to proceed or backtrack. For example, positioning the robot on a steep hill might cause it to tip over on its side, thereby incapacitating it.
2. It physically alters the robot or environment in an undesirable way. For example, a configuration may cause damage to the robot through physical contact with an object in the environment.

We assume that the above conditions for defining admissibility for a robot can be expressed as a set of N constraint inequalities of the following form:

$$f_i(p_1, p_2, \ldots, p_N) \leq K_i \quad (admissible) \tag{9}$$

$$f_i(p_1, p_2, \ldots, p_N) > K_i \quad (inadmissible)$$

where f_i is a constraint function, the P parameters are the configuration space parameters, and K_i is some threshold constant. A configuration is classified as admissible only if f_i is greater than K_i for *all* i; otherwise, the configuration is classified as inadmissible. The remainder of this section defines admissibility for the NAVLAB and describes the means for representing admissible space within the configuration space.

NAVLAB Inadmissibility

A car-like robot such as the NAVLAB propels itself by exerting a force on the ground through its wheels. The wheels must therefore remain in contact with the ground in such as way as to maintain this capability. Furthermore, as with most mobile robots, planning must ensure that the body of the robot itself does not come in contact with objects in its environment. Similar considerations were modeled in a system developed at the Hughes Corporation [12]. We begin by developing a model for the NAVLAB. We then proceed to derive constraint equations that capture the above considerations for our model. It will be shown that each constraint can be expressed as one or more constraint inequalities of the form described in Equation 9.

The NAVLAB is a complex robot. The shape is approximately box-like, but it includes a large number of small features. The ability to move forward depends on a large number of factors, including the engine specifications, the kinematic and dynamic effects of the suspension, and the friction between the wheels and the ground. In order to render our admissibility calculations tractable, we have adopted the following simplified model (see Figure 18). As explained previously, the robot's body can be modeled by a projection of all of its faces into the plane of the undercarriage, dubbed the extended undercarriage. For the NAVLAB, the extended undercarriage is rectangular. We model the NAVLAB's suspension by a spring between the undercarriage and wheel at each wheel position. Thus, the model consists of a rectangle and four connected springs. The attitude of the undercarriage is determined entirely by the geometry of the rectangle, the spring constants at each wheel, and the terrain elevation values at the four wheel positions. We define the following parameters: Let W and L be the left-to-right and front-to-back distances respectively between the wheels. Number the wheels 1 to 4 beginning with the right front

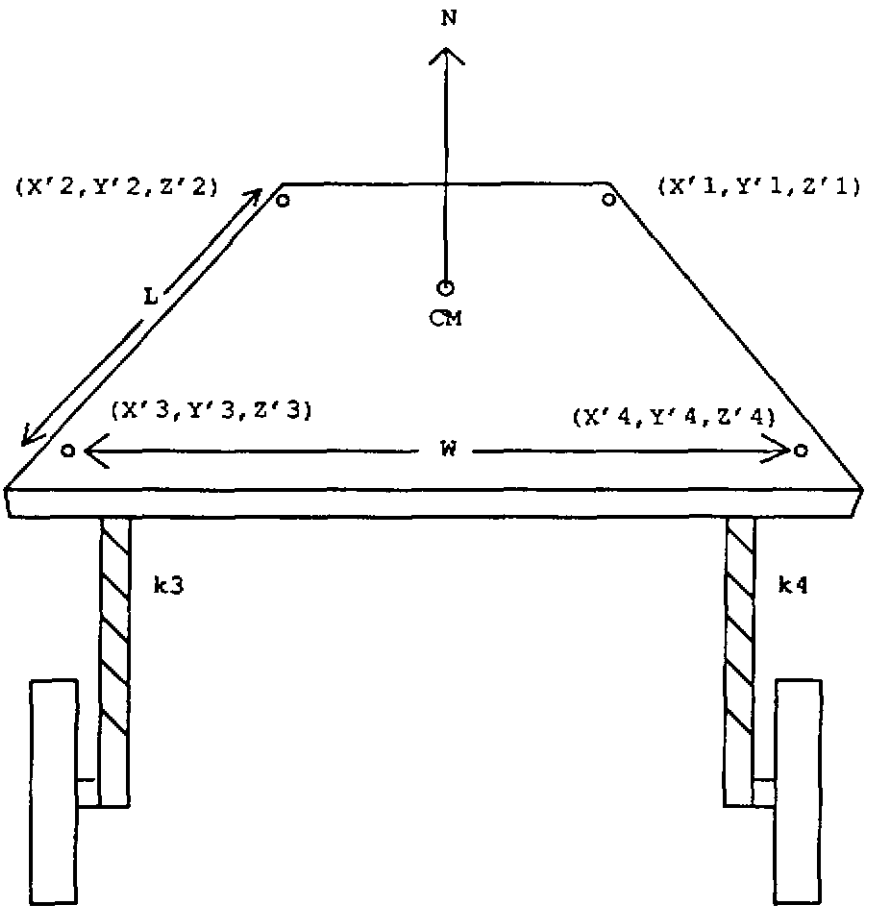


Figure 18: Model for the NAVLAB

wheel and proceeding counterclockwise as viewed from above. Let K_i and S_i be the spring constant and resting lengths respectively for the i-th wheel. Let W_r be the weight of the robot. Define a right-handed coordinate system affixed to the world with x and y in the ground plane and positive z the elevation. Likewise, define a right-handed coordinate system with x' and y' in the ground plane, positive z' the elevation, the origin of the system at the projection of the robot's center of mass into the ground plane, and positive y' aligned with the heading of the robot. Let z_i and z'_i be the terrain elevation and undercarriage elevation respectively at the i-th wheel. Let (x'_i, y'_i) be the ground plane coordinates of the i-th wheel position.

When the robot is placed on the terrain, the undercarriage assumes a position in space given by the following plane equation:

$$z' = Z_C + T_C x' + T_L y' \tag{10}$$

where T_C is the *cross-tilt*, or robot tilt per unit length from the right side to the left side of the robot, T_L is the *long-tilt*, or front to back tilt, and Z_C is the elevation of the robot's center of mass. In order to determine the three coefficients in Equation 10, we need to examine the model dynamics. Since the model remains motionless once it is placed on the terrain, the forces on and the torques about the center of mass must sum to zero, as given by the following equations:

$$\sum_{i=1}^{4} K_i(z_i - z'_i + S_i) - W_r = 0 \tag{11}$$

$$\sum_{i=1}^{4} K_i(z_i - z'_i + S_i)x'_i = 0 \tag{12}$$

$$\sum_{i=1}^{4} K_i(z_i - z'_i + S_i)y'_i = 0 \tag{13}$$

Without a loss of generality, we assume that the spring constants K_i and resting lengths S_i are equal for each wheel. For notational simplicity, we drop the subscripts on K and S. Furthermore, we assume that the robot's center of mass is in the center of the undercarriage rectangle. Thus, if we place the robot on level ground, the undercarriage will come to rest parallel to the ground, at a distance H above it. By setting H equal to $z'_i - z_i$ for all i in Equation 11, we get:

$$H = S - \frac{W_r}{4K} \tag{14}$$

Using Equation 14 and the above simplifications, we can rewrite Equations 11, 12, and 13 as:

$$\sum_{i=1}^{4} (z_i - z'_i) + 4H = 0 \tag{15}$$

$$\sum_{i=1}^{4} (z_i - z'_i + S)x'_i = 0$$

$$\sum_{i=1}^{4} (z_i - z'_i + S)y'_i = 0$$

For a given placement of the robot on the terrain, we have a total of seven unknowns: z'_1, z'_2, z'_3, z'_4, T_C, T_L, and Z_C. The four wheel points on the undercarriage plane are given by $(\frac{W}{2},\frac{L}{2},z'_1)$, $(\frac{-W}{2},\frac{L}{2},z'_2)$, $(\frac{-W}{2},\frac{-L}{2},z'_3)$, and $(\frac{W}{2},\frac{-L}{2},z'_4)$. These four points substituted into Equation 10 gives us four equations. The remaining three are provided by Equations 15. Thus, the coefficients for the plane equation 10 are:

$$T_C = \frac{(z_1+z_4) - (z_2+z_3)}{2W} \tag{16}$$

$$T_L = \frac{(z_1+z_2) - (z_3+z_4)}{2L}$$

$$Z_C = \frac{(z_1 + z_2 + z_3 + z_4)}{4} + H$$

On rough terrain, it is possible for the robot to encounter terrain sloped enough to cause tipping. Let Z_{CM} be the vertical distance from the wheel axles to the robot's center of mass on level ground. The robot will tip when the deviation from horizontal exceeds the point where the plane defined by the center of mass and the two wheels with lowest elevation is vertical. The maximum angular deviation from horizontal permitted without tipping in the long and cross directions is given by:

$$\phi_L = arctan(\frac{2Z_{CM}}{L}) \tag{17}$$

$$\phi_C = arctan(\frac{2Z_{CM}}{W})$$

T_L and T_C are the sines of the angles of the body's deviation from the horizontal in the long and cross directions respectively. Let T_{C0} and T_{L0} be the sines of ϕ_C and ϕ_L respectively. A configuration is classified as admissible only if both of the following constraints hold:

$$|T_L| \leq T_{L0} \tag{18}$$

$$|T_C| \leq T_{C0}$$

In addition to tipping, rough terrain can give rise to situations where the wheels cannot deliver power to the ground, usually resulting in wheel slippage. Slippage can occur due to a number of reasons, the two most important being a low effective coefficient of friction between the wheels and the ground and situations where an active wheel must climb the terrain (e.g., out of a ditch or up a grade).

We assume that the coefficient of kinetic friction is bounded on the low end by μ_k, and that the maximum difference in elevation in the vicinity of a given active wheel is given by Δz. The most pathological situation given these parameters is illustrated in Figure 19. In this situation the wheel supports the robot at the high corner of the "step", where the frictional support is minimal. If the weight of the robot on this wheel exceeds the frictional force, the robot will slip and thus, will be unable to climb the step. In this figure, r_w is the radius of the wheel, and ϕ is the angle between the gravity vector and the tangent line to the wheel at the step's corner. The force at this point in the direction of ϕ due to friction is given by:

$$F_{f,\phi} = \mu_k F_w \sin(\phi) \tag{19}$$

where F_w is the force on the wheel due to the robot's weight. The force needed to support the robot's weight at the corner along the direction of ϕ is given by:

$$F_{w,\phi} = F_w \cos(\phi) \tag{20}$$

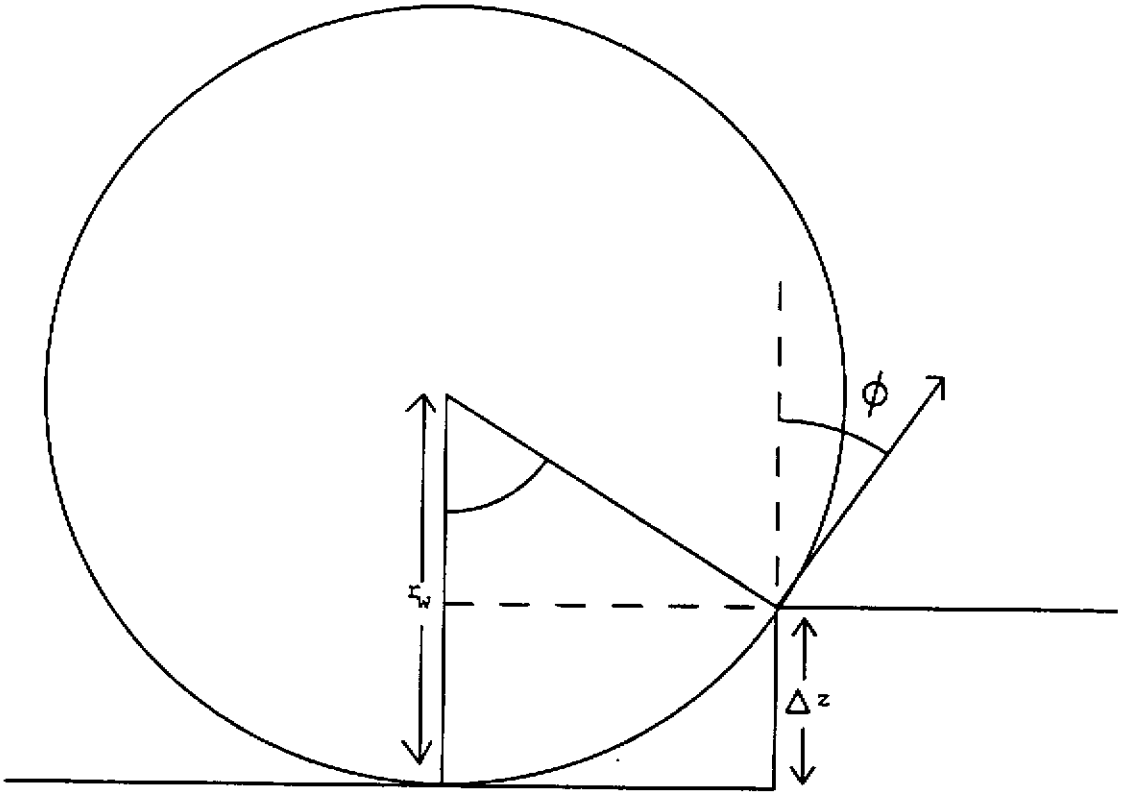


Figure 19: Conditions for Wheel Support

Therefore, in order prevent slippage, $F_{f,\phi}$ must be greater than or equal to $F_{w,\phi}$, or:

$$\mu_k F_w \sin(\phi) \geq F_w \cos(\phi) \tag{21}$$

From Figure 19, we have:

$$\sin(\phi) = \frac{r_w - \Delta z}{r_w} \tag{22}$$

$$\cos(\phi) = \frac{\sqrt{2\Delta z r_w - \Delta z^2}}{r_w}$$

Substituting Equations 22 into Equation 21 gives:

$$\frac{r_w - \Delta z}{\sqrt{2\Delta z r_w - \Delta z^2}} \geq \frac{1}{\mu_k} \tag{23}$$

Squaring the above equation, solving the quadratic form, and discarding the meaningless root gives:

$$\Delta z \leq r_w - \frac{r_w}{\sqrt{\mu_k^2 + 1}} \tag{24}$$

The intuition behind Equation 24 is that any "steps" in the terrain must be less than the radius of the wheel. Exactly how much less is determined by the effective coefficient of friction. The smaller the coefficient of friction, the smaller the step that can be negotiated. In the extreme case with no friction (i.e., $\mu_k = 0$), no difference in terrain elevation can be tolerated, (i.e. $\Delta z = 0$).

Attempting to measure μ_k and substituting the result into Equation 24 is inappropriate. The interaction between a real robot on real terrain involves much more than just the coefficient of friction. Factors such as elasticity of the tires and terrain, soil characteristics, air pressure in the tires, side-wall support contribute as well [3, 4]. Rather, it is important to note that the constraint can be formulated as a maximum step size (Z_w), the actual value of which can be determined experimentally using a real robot on real terrain. Taking all four wheels together (both active and passive), the resultant constraint is a function of Z_w for each wheel along with the long- and cross-tilt of the body. The analytic form is quite complex and stretches the usefulness of our simple model. Alternatively, we propose an empirical approach where thresholds are experimentally determined for the active and passive wheels parameterized by the tilt values. For our tests, we assumed no significant tilt and imposed maximum thresholds on each wheel separately of the form:

$$\Delta z \leq Z_w \tag{25}$$

As described above, the robot's body is modeled by a projection of all of its faces into the plane of the undercarriage. We need only check the intersection of this extended polygon with the terrain for body collisions. The plane of the undercarriage z' is given by Equation 10. The body clearance constraint can be written as follows:

$$\forall (x,y) \in S, f_t(x,y) < z'(x',y') \tag{26}$$

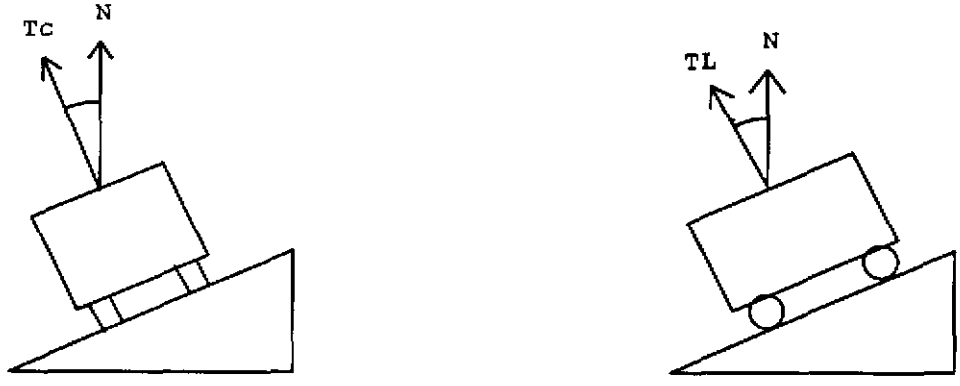
S is the set of coordinates (x,y) in world coordinates that fall beneath the extended rectangle. The coordinates (x',y') are the corresponding coordinates in the robot frame. The function f_t is the *terrain function*, that is, it specifies the terrain elevation for a given xy-coordinate point. The above constraint states that for a given configuration to be admissible, all portions of the terrain that fall beneath the robot must be lower than the plane of the undercarriage.

The attitude, support, and clearance constraints given by Equations 18, 25, and 26 are illustrated in Figure 20.

The Constraint Solution Space

As the robot moves from a starting configuration to a goal configuration, it passes through a continuum of configurations. This continuum must be tested for admissibility. In the Hughes system [12], candidate trajectories are approximated by a sequence of configuration points, and each configuration point is individually tested for admissibility. In order to minimize the chance that a trajectory passes through an inadmissible configuration, the sequence must be closely-spaced. Let ε_s be the spacing between adjacent configuration points in each dimension (assumed equal for the sake of simplicity). If each dimension of the planning space is of size p_s, then the number of divisions along each dimension is $d_s = \frac{p_s}{\varepsilon_s}$. Since the number of degrees of freedom for the NAVLAB is 3, the maximum number of configurations that must be tested to find a path (worst case) is:

$$O(d_s^3) \tag{27}$$



Tilt Constraint

Support Constraint

Body Clearance Constraint

Figure 20: Admissibility Tests for the NAVLAB

Since a large number of tests must be performed, the number of candidate trajectories considered must be necessarily restricted. If the robot could classify a large subspace of configurations as admissible in one operation, planning through this subspace could proceed quickly. Likewise, if it could classify a large subspace as inadmissible, it wouldn't waste computation time attempting to plan through it.

Consider a box-shaped subspace of configurations (known as a *voxel*) bounded in three dimensions by $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$, and $[\theta_{min}, \theta_{max}]$. We would like to be able to determine if all of the configurations in the voxel are safe, if all are unsafe, or if the voxel contains a mixture of both safe and unsafe configurations. Another way of stating this is whether the voxel lies entirely inside the solution space for

the constraint inequalities, entirely outside of this space, or if it straddles one or more boundaries.

In order to classify a particular voxel as safe (admissible), we need to show that for each constraint inequality, f_i, the smallest value for f_i with parameters p ranging over the voxel is greater than K_i. In order to classify a voxel as unsafe (inadmissible), we need to show that the largest value is less than or equal to K_i for at least one constraint, f_i. If the constant K_i splits the range for at least one f_i, then the voxel contains a mixture.

One way to do this is to differentiate each function f_i with respect to the planning parameters p and examine all local maxima (or minima) corresponding to sets of p that fall within the bounds of the voxel. Because we are examining f_i over only a subset of its domain, the maximum (or minimum) of f_i in the voxel may not occur at a local maximum (or minimum), but instead may occur on the border of the voxel. For this reason, the values of f_i along the faces of the voxel must also be examined.

The above technique is tedious and potentially expensive computationally, particularly if the number of configuration space parameters is large, or if the constraint equations are complex. Instead of attempting to compute the least upper bound (LUB) or greatest lower bound (GLB) analytically for f_i, we show below that it is much simpler to compute some upper bound (UB) and some lower bound (LB), such that $UB \geq LUB$ and $LB \leq GLB$. If UB and LB are used to classify voxels, the algorithm is sound in labelling voxels as admissible or inadmissible, but sometimes incorrectly labels admissible or inadmissible voxels as both (a mixture). For reasons explained later in this section, this error is acceptable. We have adopted a technique known as the SUP-INF method [5] for determining the bounds UB and LB. This technique was originally developed for handling universally quantified linear constraints. It was extended in ACRONYM [6] to handle a class of nonlinear constraints as well, for purposes of predicting the appearance of a three-dimensional object in an image. We have extended it to include a larger class of nonlinear constraints and have used it to test the intersection of a voxel with the solution space of a set of constraints. The technique works as follows: The upper and lower bounds (UB and LB) are computed in the following manner. The constraint function f_i is treated as a composite function of *primitive* functions $g_1, g_2, \ldots, g_N$. Examples of primitive functions used are addition, subtraction, multiplication, division, absolute value, sine, cosine, arctangent, etc. The function f_i can be viewed as a tree where the vertices are primitive functions and the leaves are the planning space parameters, as illustrated in Figure 21. Lines emanating down from a vertex are input parameters to the primitive function and the line emanating up is the value returned.

The upper and lower bounds (UB and LB) for f_i are computed by inserting the maximum and minimum values of the planning space parameters over the voxel under consideration into the leaves of the tree and propagating the max and min values to the root. The problem is thus reduced to determining the LUB and GLB over some domain for a restricted set of primitive functions. The primitive functions are simple enough such that this is not a difficult task.

Since all three constraints we are considering depend on the positions of the four wheels, for a given voxel we need to bound their positions. The transformation equations from the robot coordinate frame to the world coordinate frame as a function of a configuration (x_c, y_c, θ_c) are:

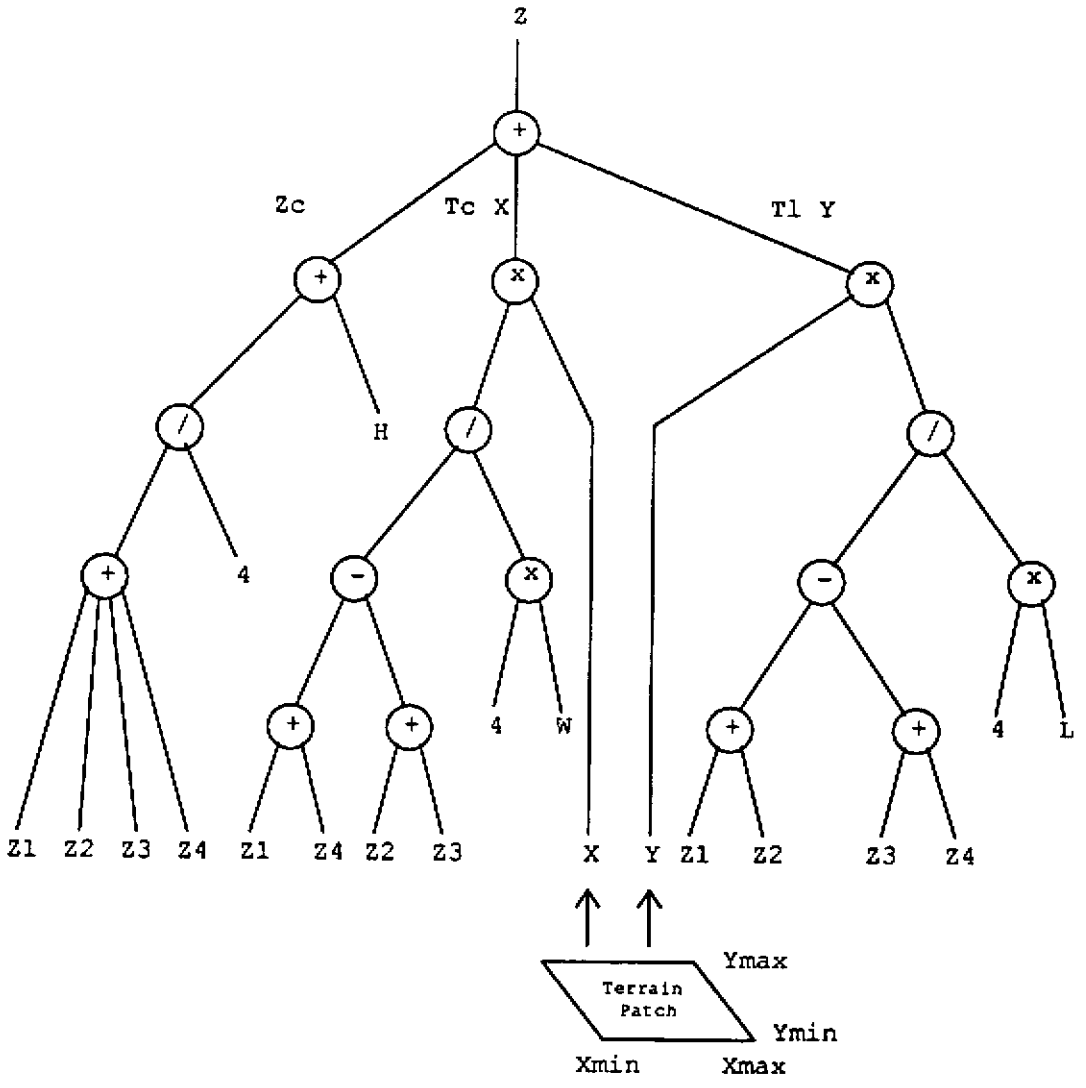


Figure 21: Evaluating the Body Clearance Constraint for Admissibility

$$x = x' \cos(\theta_c) - y' \sin(\theta_c) + x_c \tag{28}$$

$$y = x' \sin(\theta_c) + y' \cos(\theta_c) + y_c$$

Using the SUP-INF technique, we substitute the four wheel positions (x'_i, y'_i) into the above equations to compute bounding boxes in x and y for each wheel. The terrain function, f_t, is examined to compute the UB and LB on z_i for each wheel. These bounds are subsequently inserted into Equations 16 to determine bounds on the plane coefficients T_L, T_C, and Z_C. The bounds on T_L and T_C are used to evaluate the robot attitude constraint (Equations 18). The *ranges* of the z-values at the four wheels are used to evaluate the suspension support constraint (Equation 25).

The body clearance constraint is evaluated by comparing the bounds on the plane equation with the terrain that falls beneath the robot. For each patch of terrain which lies beneath the undercarriage, we need to calculate the UB and LB for the clearance constraint function across the wheel and terrain patch parameters. If the LB is greater than the maximum z for *all* terrain patches under consideration, the voxel is labelled as admissible. If the UB is less than the minimum z value for a *single* terrain patch, the voxel is labelled as inadmissible. Otherwise, the voxel is labelled as both (a mixture).

Figure 21 shows the propagation tree for computing the UB and LB for the body clearance constraint for a terrain patch beneath the robot. Note that the inputs to the constraint function are the bounds on the four z values for the wheels as well as the x and y bounds for the terrain patch. The result (UB and LB) are checked against the max and min z values for the terrain patch.

Considering all constraints together, a voxel is labelled as admissible only if all of the constraints are admissible. It is labelled as inadmissible if at least one constraint is inadmissible. Otherwise, it is labelled as both.

The Octree

The algorithm in the preceding section provides a means for classifying a box-shaped voxel as admissible, inadmissible, or both. The actual shape of the admissible (or inadmissible) subspace within a configuration space may be far from box-shaped. One solution is to approximate the admissible space with a large number of small boxes. As explained in the previous section, this solution is expensive both in memory and computation time. If the admissible space tends to be concentrated in certain areas, then large boxes can be used to represent this area, and smaller boxes can be used to represent the fragmented areas. Such a representation is known as an *octree*. The octree was first used in CSG systems [31] for approximating the shape of solid objects. Later in planning, octrees were used for representing admissible space for a manipulator [17] and a mobile robot [18] moving about in a field of polyhedral obstacles. We have utilized this data structure in conjunction with the SUP-INF technique discussed in the previous section in a general and efficient framework for representing satisfaction (admissibility) for any set of constraint inequalities.

In order to construct an octree, we begin by testing the admissibility of a large voxel that bounds the entire configuration space. If the voxel is either admissible or inadmissible, the construction stops. If the voxel is classified as both, it is split into eight subvoxels and the process recurs on each subvoxel. Recursion terminates along any given branch when all "both" nodes have been resolved or when a maximum depth (resolution) in the tree has been reached. Back pointers to the parent voxel are maintained for each subvoxel such that the final data structure is a tree, where the leaf nodes are labelled as admissible or inadmissible, and the vertices are labelled as both. Any nodes not classified as both at the maximum resolution are labelled as inadmissible and the recursion terminates.

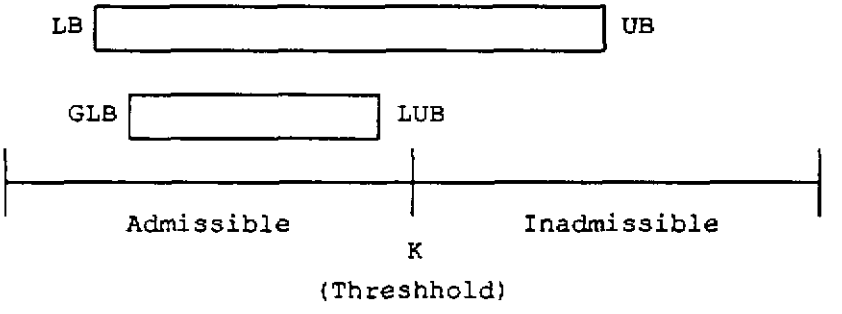


Figure 22: Misclassification Due to Conservative Bounds

Note that in the process of expanding a "both" node, we may discover that it is actually a misclassified admissible or inadmissible node, due to the fact that we are using *UB* and *LB* rather than *LUB* and *GLB* in the classification process. This case is illustrated in Figure 22. In this example, a voxel is classified as both even though it is admissible because the *UB/LB* interval straddles the threshold, while the *LUB/GLB* interval does not. If all of the sons of a "both" node are discovered to be admissible (or inadmissible), we can reclassify the parent as admissible (or inadmissible) and propagate the change up to the root.

The principal advantage of the octree is that it is an efficient way to represent clusters of admissibility (or inadmissibility) in the planning space. This is often the case when the local environment includes such inadmissible structures as trees, wide ditches, broad and steep slopes, etc. As shown in Equation 27, if we evaluate the admissibility of the planning space at each configuration, the total number of operations required to classify the whole space is cubic in number of discretizations along each dimension. Assuming that voxels containing only admissible or inadmissible configurations are never subdivided, the total number of operations required to evaluate the admissibility of the space is roughly proportional to the number of configuration points on the surface of the boundary between admissible and inadmissible space. Thus, the octree is a major improvement over the dense representation, since high-resolution voxels are needed only to represent the obstacle boundaries (two-dimensional) rather than the entire planning space (three-dimensional).

Figures 23, 24, and 25 illustrate octree constructions for the tilt, support, and body collision constraints respectively. These figures depict a cross section of the octree in x and y for θ ranging from 0 to 1.25 degrees. The scale of the vehicle is shown in all three figures. The configuration parameters (x, y) specify the position in the ground plane of the middle of the vehicle's rear axle while θ specifies the heading as measured from the positive y-axis. The crossed and open boxes represent inadmissible and admissible regions respectively. In all three figures, the environment is approximately flat except for the square drawn with thick lines. In Figure 23, the square slopes upward along the positive y-axis (vertical) at 45 degrees, 15 degrees greater than the safe threshold. Note that the high resolution voxels define the boundary between the admissible and inadmissible space. Along the sides of the tilted square, the boundary corresponds to those configurations that place two wheels on the edge of the square. Along

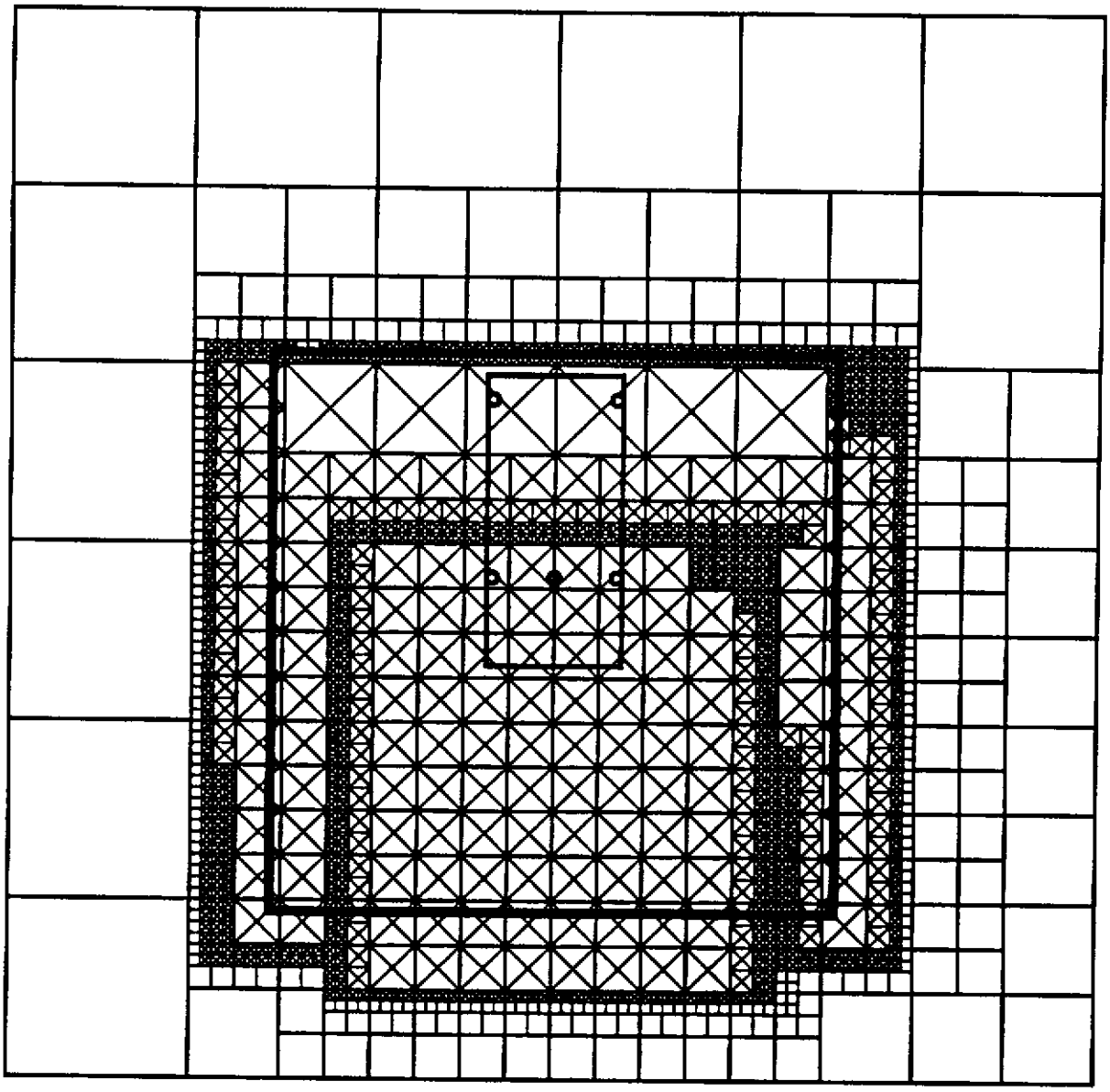


Figure 23: Octree Slice for the Tilt Constraint

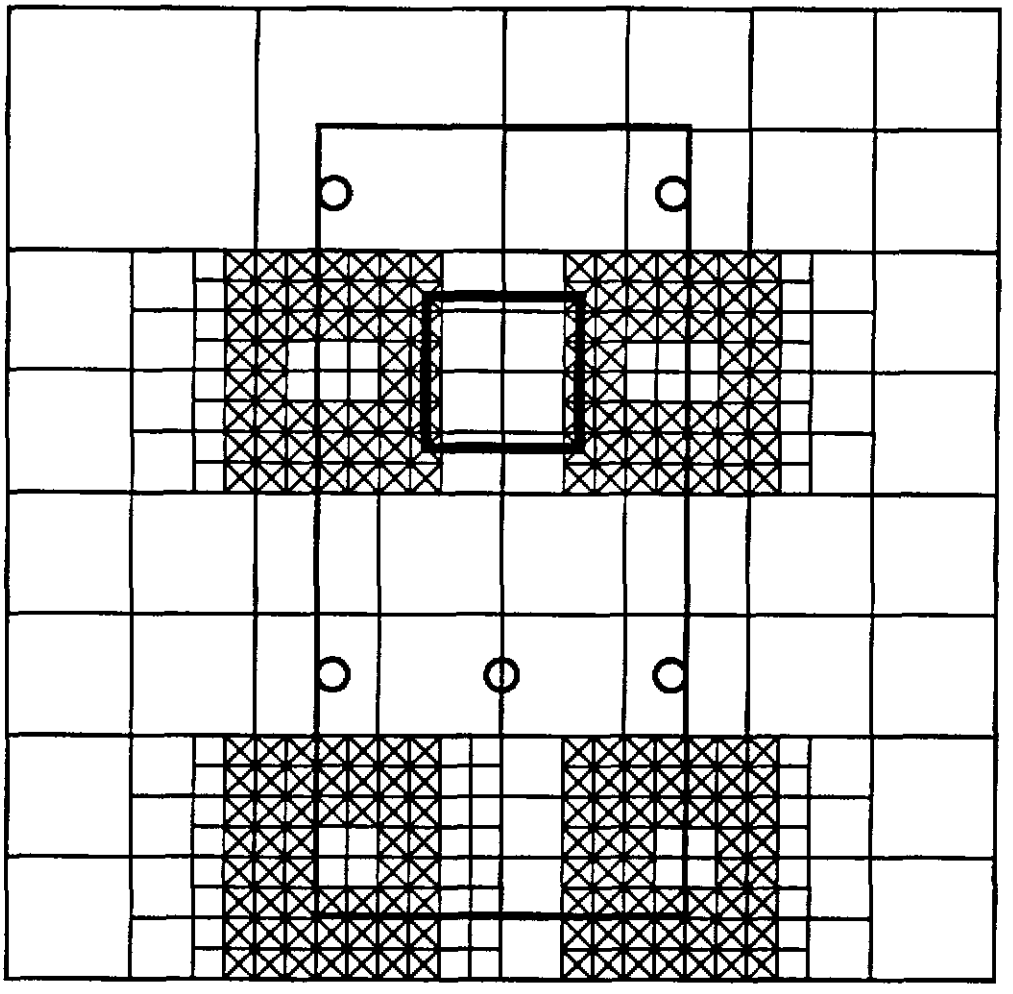


Figure 24: Octree Slice for the Support Constraint

the front the boundary corresponds to those configurations where the front wheels have climbed to a high enough elevation to exceed the maximum tilt.

In Figure 24, the polygon is 40 cm high, exceeding the maximum step by 10 cm. Note that four inadmissible regions result, corresponding to the configurations that place each of the four wheels against the front of the polygon. Note also that the configurations that place the wheel atop the square are admissible (although not reachable). In Figure 25, the polygon is 1 meter high, 50 cm higher than the undercarriage. Configurations that bring the body in contact with this square are classified as inadmissible.

In all three of these figures, note that many voxels containing only admissible or only inadmissible configurations are subdivided. This effect arises from the use of conservative bounds on the constraint functions. In these cases, some efficiency in representation is traded for fast computation of constraint bounds.

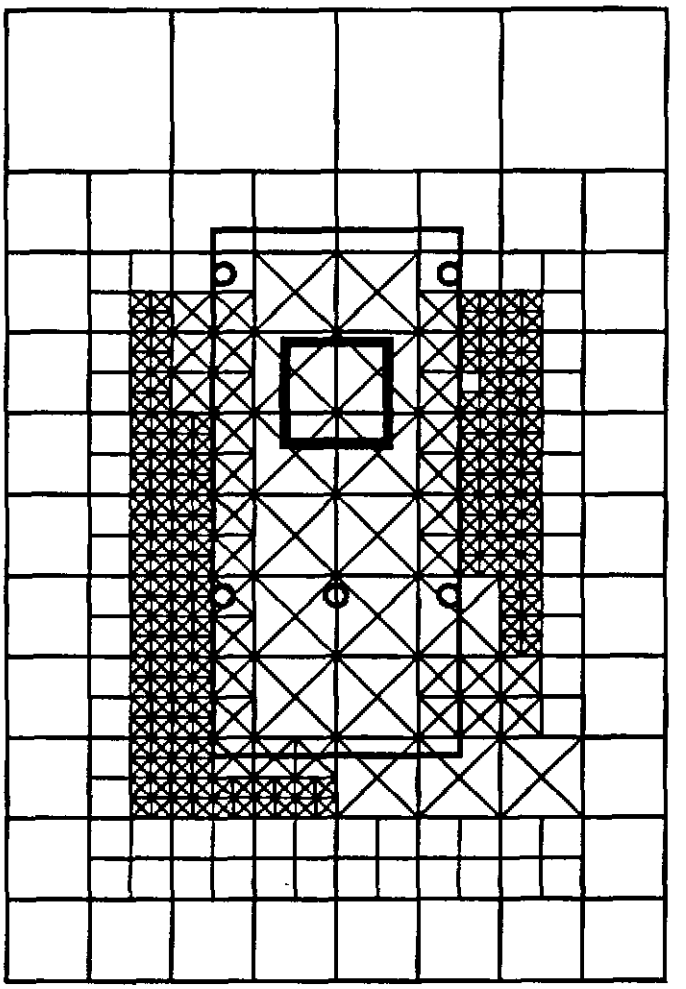


Figure 25: Octree Slice for the Body Collision Constraint

The Terrain Function

As described in the previous section, admissibility for a mobile robot can be expressed as a set of constraint inequalities defined over the configuration space parameters. Admissibility is computed using the SUP-INF algorithm over these constraint functions. These constraint functions are composed of a set of primitive functions, one of which is the *terrain* itself. The terrain can be viewed as a function:

$$z = f_t(x, y) \tag{29}$$

returning an elevation z value for a given pair of ground coordinates (x, y). Since information about the terrain is generally acquired in a pixel-by-pixel fashion spanning a grid, we might be tempted to represent f_t as a lookup table. Such a representation renders operations such as computing upper and lower bounds on the terrain elevation across an arbitrary rectangle computationally expensive. Ideally, we

would like to have a simple, analytic function for approximating the terrain. Simple functions, however, tend to approximate unstructured terrain poorly, while complicated functions are difficult to bound over an arbitrary interval.

Our approach is to represent bounds on the terrain elevation in a coarse-to-fine manner using a terrain pyramid. The coarse data in the pyramid expedites queries by eliminating the need to inspect the terrain at a high resolution everywhere. The advantage of this approach is that it does not require that the terrain be structured.

Terrain Preprocessing

The NAVLAB acquires three-dimensional terrain data about its environment through the use of an ERIM laser rangefinder. The rangefinder scans an area about eight meters wide and four meters deep out to six meters in front of the robot. The range information is recorded in a sensor-based, polar coordinate frame. This range map is converted to a robot-independent coordinate frame by retaining the maximum elevation (z value) at each world point (x,y). This elevation map can then be indexed independently of the robot's position.

Because the communication links in CODGER are not of sufficient bandwidth to permit the transmission of images between Perception and Planning modules, the elevation map is converted to a polygonal mesh to reduce the size of the data. The mesh is generated by running a Canny operator [8] over the data to detect discontinuities in curvature. A connected components algorithm identifies the smooth regions between the discontinuities. Polygons are fit to the segmented regions and a plane is fit to the surface enclosed by each polygon. Figure 40 shows a sequence of three polygonal meshes for real off-road data. See Hebert [27, 28, 29] for details.

Since all of the information about the terrain is captured in the positions, orientations, and boundaries of the polygons (numbering 5 to 40 for typical terrain), the size of the data is greatly reduced. Furthermore, this preprocessing is useful for eliminating for noise-filtering. For example, it can be used to eliminate spurious pixels. Unfortunately, the time required to construct the necessary data structures for trajectory planning from this mesh can be quite high, such that future systems will circumvent the mesh and transmit elevation maps directly over a high-bandwidth channel.

The Terrain Pyramid

As previously described, the robot needs to index into its local environment to determine whether a given voxel of configurations is admissible. In order to make this determination, the robot needs the maximum and minimum terrain elevation values for each wheel "box" in order to compute the range of positions of the undercarriage plane. Furthermore, the terrain beneath the undercarriage must be compared to the undercarriage plane for intersection. If we use the iconic (pixel) level representation for the terrain function, this last operation amounts to convolving the robot's undercarriage with the terrain over the extent of the voxel using a pixel-to-pixel test for intersection. Thus, the complexity is a linear function of the number of pixels spanned by the undercarriage and is quite high.

To circumvent this problem, we have employed a terrain pyramid. The pyramid is illustrated in Figure 26. The bottom level is the iconic range image data (elevation map). Each terrain node corresponds to a single pixel and holds the elevation value for that pixel. The level above it reduces the image data by a factor of two in each dimension. Each terrain node holds the maximum and minimum elevation values for the four corresponding sons below it. The pyramid is constructed recursively from the bottom level to the

top. The top level consists of a single terrain node holding the maximum and minimum elevation values for the entire environment. The algorithm for computing the maximum and minimum terrain elevation values for a given query rectangle R is given below. Initially, T bounds the local environment and z_{max} and z_{min} are set to NULL.

1. If the terrain patch T lies entirely outside of rectangle R, return.
2. If T lies entirely inside of R update z_{max} and z_{min} and return.
3. T intersects R (boundary overlap). If T is not a leaf (i.e., not in the bottom level), divide T into four quadrants and recur at Step 1 with R and each T_i. Return.
4. T is a leaf. Update z_{max} and z_{min} and return.

The terrain pyramid can be constructed with fewer operations than twice the number of pixels in the local environment. The complexity of a query for a rectangle R is roughly proportional to the number of terrain nodes of maximum resolution that fall along the perimeter of R. Figure 26 illustrates the terrain nodes of different resolutions bounded by a typical query rectangle.

The terrain pyramid is a powerful representation because it assumes no underlying structure in the terrain. It can be constructed efficiently and rectangle queries can be processed efficiently, short of examining all pixel values interior to the rectangle. The time needed to process a query is roughly proportional to the size of the perimeter of the rectangle, rather than the area. Currently, the pyramid is constructed from the polygonal mesh. Future systems will construct it directly from the elevation map.

The Planning Paradigms

It is the goal of a path planner to move a robot from some starting configuration to a goal configuration while passing through only admissible configurations. We have identified five criteria for evaluating path planning algorithms:

- *Admissibility Model:* all planners require a space of admissible configurations in which to search for robot trajectories. This space can be constructed a priori or as the robot plans. The construction process itself can range from simple to intractable.
- *Soundness:* a sound path planner is guaranteed to find a trajectory through the search space that can be executed by the robot. More specifically, it means that the trajectory passes through admissible configurations only, even in the presence of uncertainty in the robot's control and environment. Furthermore, the robot is able to execute the trajectory, given its kinematic constraints.
- *Completeness:* a complete path planner is guaranteed to find a trajectory through the search space from start to goal, provided one exists. If no such trajectory exists, the planner is able to detect and report this condition.
- *Optimality:* an optimal path planner is guaranteed to find the optimal path (based on some path measure) within the search space.

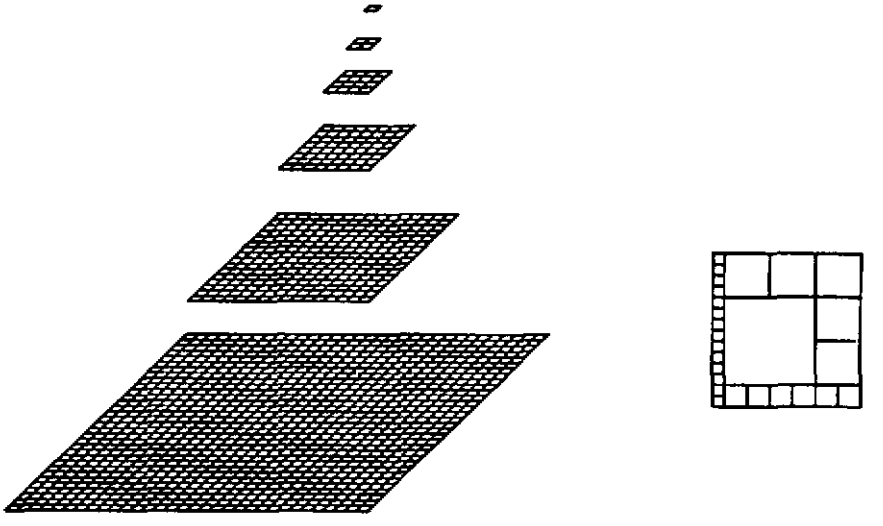


Figure 26: The Terrain Pyramid

- *Complexity:* the running time of a planning algorithm can be a function of the size of the planning space, the number of constraints in the environment, or the maximum allowable deviation from optimality.

A number of paradigms exist for path planning. In the following sections, we briefly describe these paradigms and evaluate them based on the above five criteria.

Vertex-Graph Planning

Vertex-graph planning [36, 38] was developed to plan trajectories for a circular robot across a smooth, flat surface through a field of polygonal obstacles. The search space consists of a graph in which the nodes are obstacle vertices "expanded" by the radius of the robot and the arcs are straight-line paths connecting vertices such that the line segments do not intersect with other obstacles. An A* search is used to find the optimal path. The complexity of the algorithm to construct the graph is $O(N^3)$, where N is the number of vertices in the environment. A system was developed at the Jet Propulsion Laboratory [49] that reduces the time required to construct the space without sacrificing optimality by constructing the graph as the robot plans. In a later system [37], the planning algorithm was extended to handle a polygonally-shaped robot. Polyhedral approximations to the inadmissible space were employed at only a slight cost in optimality. In [47], an algorithm was developed to solve the *generalized piano movers problem*. In this problem, the robot is modeled by a set of polyhedra. It moves about in a field of polyhedral obstacles. The complexity of the algorithm is the product of the number of constraints in the robot and the environment (i.e., vertices and edges in the polyhedra) all raised to a double-exponential in the number of degrees of freedom of the robot, thus making it prohibitively expensive. More recently, an algorithm was developed [9] that reduces the complexity to single-exponential.

The advantages of the V-graph approach are: first, the paths it generates are optimal; second, the approach is also complete, i.e. if a path exists it will be found. The disadvantages are numerous. First,

the paradigm assumes that the search space can be constructed and represented analytically. This is not difficult for the "traditional" planning domain, where a circular robot moves about on a flat surface between polygonal obstacles. In more complicated domains, such as that of the NAVLAB, inadmissibility is defined by a large number of constraint inequalities defined across the planning space parameters. Solving for the boundary between admissible and inadmissible space is difficult in such a general case. Furthermore, the definition of "vertices" on the boundary becomes much more complicated. Second, the V-graph approach takes the robot as near as possible to the obstacles. Any control error whatsoever could result in a collision. Third, the resultant paths consist of a functional form that the robot may not be able to execute. In many of these planners, the trajectory is a piece-wise sequence of line segments through the planning space. Robots such as the NAVLAB, which cannot turn in place, cannot execute such a path. Fourth, the complexity of these algorithms is strongly tied to the number of constraints in the environment. For highly-convoluted terrain, this number can be large.

Free-Space Planning

Free-space planning techniques [7, 20] construct a search space of the free (admissible) regions in the robot's space, rather than the obstacle vertices. The nodes in the graph consist of the free-space regions and the arcs are unobstructed boundaries between these regions. A* is also used to find the "best" path. The space construction complexity is also $O(N^3)$, where N is the number of free-space polygon vertices.

The advantage of the free-space approach is that by modeling the free regions, the planner can move the robot through the center of the region, thus reducing the chance of collision due to control error. The free-space approach is also complete. The disadvantages of this approach are numerous. First, by moving the robot through the centers of the regions, the planner can generate grossly suboptimal paths, especially when planning through large regions. Second, this approach also requires that the boundaries of inadmissible space be computed analytically. Third, the resultant paths may also be unsound, especially through narrow free-space regions. Fourth, the complexity is also high for "difficult" environments.

Tesselation Planning

In the tesselation approach to planning, the search space is decomposed into regular-sized subspaces (such as squares or voxels). The nodes in the search graph are the units containing only admissible configurations. Two nodes are connected with an arc if they share a boundary. In some recursive schemes [17], the boxes can be decomposed into smaller sizes for finer planning resolution. The space construction complexity is $O(N^r)$, where N is the number of tesselations along each of r degrees of freedom. Search algorithms include A* [50] or gradient descent (potential field) [32, 34] approaches.

The overriding advantage of the tesselation approach is that the admissible/inadmissible boundaries need not be computed analytically. Instead, the algorithm need only determine whether or not the boundary passes through a particular voxel in space, and if not, whether the voxel lies entirely inside or outside of the admissible space. This capability allows us to construct a voxel-based approximation of the inadmissible space, regardless of its analytic form. Second, the paths are optimal within the resolution of the tesselation if A* is used. If gradient descent is used, the path generation is fast but neither optimal nor complete (since the trajectory can get caught in a local minima--these schemes do not backtrack). In a later approach [30], gradient descent was combined with A* to recover from local minima. This technique preserves completeness at the expense of a greater sacrifice in optimality. The disadvantage is that existing approaches do not guarantee path soundness. There is no guarantee that the robot will

be able to move from node to node in the space. Typically, the robot is assumed to be omnidirectional in these approaches.

Planning for Outdoor Mobile Robots

For robots such as the NAVLAB that operate in unstructured environments, it is virtually impossible to construct the boundaries between admissible and inadmissible space analytically. As explained in previous section, admissible space can be defined by a number of arbitrarily complex equations. Therefore, it is of paramount importance that a planning scheme for these robots allow approximations to these boundaries that are computationally *easy* to construct. Concerning soundness, complex robots such as those with wheels, treads, or legs are typically not omnidirectional. It is therefore important that a planning scheme for these robots generates admissible trajectories. Concerning completeness and optimality, the plans generated by these robots are piece-wise trajectories of a larger, global plan. Typically, this plan is somewhat coarse. The additional work expended by the robot in executing slightly suboptimal trajectories at the local level is small compared to the cost inherent in the "coarseness" of the global path itself. Therefore, optimality is the least important of the five criteria. Completeness is more important, since failing to find a path through the local search space means that the global plan needs to be adjusted. Concerning complexity, of course we would like the fastest algorithm. Since optimality is not that important, in most cases (e.g., uncluttered environments) the resolution (voxel size) at which we plan need not be very high. Thus, the total number of voxels in the planning space need not be very large. Due to the unstructured nature of the terrain, however, the number of environmental constraints on the robot can be large. Therefore, it makes sense to choose an algorithm with a complexity that is a function of the planning resolution, rather than the number of constraints (assuming all else equal).

For these reasons we have adopted the tesselation paradigm. In the following sections we describe a new algorithm for integrating kinematic soundness into this paradigm. The merits and drawbacks of this approach are discussed as it is developed.

Kinematic soundness can be stated as follows: The robot resides at some configuration point $(p_1, p_2, ..., p_N)$ in the planning space. It is moved by applying a control vector $(c_1, c_2, ..., c_M)$ to the configuration point to move it to another configuration point. There may be restrictions on the allowable control such that it is not possible to reach an arbitrary configuration point from a given one. For example, a robot such as the NAVLAB cannot slide sideways along the ground. It is capable only of moving in the direction that its front wheels are pointing. The NAVLAB is controlled by specifying a sequence of arcs, each parameterized by curvature k and length d. Since the NAVLAB is a car-like robot, it has a minimum turning radius. Thus, the allowable curvatures are bounded by k_{min} and k_{max}, where $k_{min} = -k_{max}$.

Modeling Kinematic Constraints for Planning

The basic approach of tesselation-based planning is to divide the configuration space into subspaces called *voxels* and to search for the "best" trajectory that passes through admissible voxels. Rather than consider trajectories directly, sequences of voxels from start to goal are examined. If a given trajectory can be characterized by a unique sequence of voxels through this space, if this trajectory can be derived from the sequence, and if all possible sequences of voxels are examined by the planner if need be, then the planner is both sound and complete. Provided the right search algorithm is used, the planner is optimal to the resolution of the voxel size. Existing approaches search over sequences of voxels, assuming that an admissible trajectory exists within any given sequence. For this reason, these planners

are neither sound nor complete.

In this section, we describe a new technique for determining whether an admissible trajectory exists within a given voxel sequence to ensure both soundness and completeness. Searching the voxels for an admissible sequence from start to goal is the topic of the next section.

Representing Trajectories with Voxel Sequences

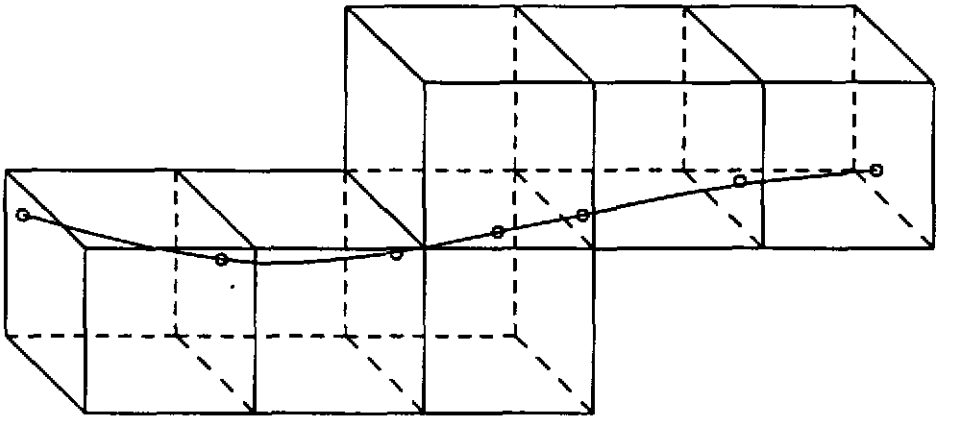


Figure 27: Sequence of Voxels Bounding a Trajectory

Consider a configuration space (x, y, θ) through which some path (trajectory) exists (at least one) from a starting configuration to a goal configuration. Consider also a tesselation of this configuration space into voxels (for argument's sake, assume they are uniform-sized). The solution trajectory is a curve in three-space that moves from a start point to a goal point through voxels in this configuration space. Since the voxels do not overlap, the trajectory is entirely bounded by a unique sequence of voxels $(v_1, v_2, ..., v_N)$, such that adjacent voxels in this sequence share a face in the configuration space (see Figure 27). For a given voxel v_i in the sequence, the trajectory enters the voxel through the face shared with voxel v_{i-1}, travels entirely within voxel v_i, and exits through the face shared with voxel v_{i+1}.

The local path planning problem does not begin with the solution trajectory, of course. In tesselation-based planning, possible sequences of admissible voxels from start to goal are considered. A sequence of voxels is considered admissible if at least *one* legal trajectory exists through it from the first voxel to the last. Assuming that the planner eventually considers a sequence of voxels containing a solution trajectory, and assuming this trajectory can be derived from the sequence, then the planner is *complete*, that is, if a path exists from start to goal the planner will discover it.

Given a sequence of admissible voxels through a configuration space, it might not be possible to find an *admissible* trajectory through these voxels. An admissible trajectory is one that is *sound*, that is, the robot is able to execute it. Most robots are not omnidirectional: they cannot move from a starting configuration to any goal configuration along a straight line *in configuration space*. Therefore, for a given voxel v_i, there may exist pairs of configuration points (c_r, c_s), with c_r on the face adjacent to voxel v_{i-1} and c_s on the face adjacent to v_{i+1}, such that no admissible trajectory exists between them lying entirely within v_i.

Assume we are given a sequence of configuration points $(c_1, c_2, ..., c_N)$ that define the "puncture points" through which the trajectory passes in the faces connecting the N adjacent voxels. If we could determine whether an admissible trajectory exists that passes through these N configuration points, then the problem of determining whether or not an admissible trajectory exists for the sequence of voxels is reduced to one of searching across all possible sequences of puncture points.

Furthermore, if we assume that the robot can be controlled independently at any point along its trajectory, that is, control decisions in the i-th voxel do not depend on those in the $(i+1)$-th voxel except through the configuration point connecting them, then the problem of determining whether an admissible trajectory exists through a sequence of puncture points can be reduced to one of finding the $N-1$ admissible subtrajectories between adjacent puncture points.

Sound Trajectories for the NAVLAB

As explained previously, the NAVLAB moves about by executing a sequence of arc trajectories of lengths d_i and curvatures k_i, such that $|k_i| \leq k_{max}$ for each k_i, where k_{max} is the maximum curvature. Since the robot can be controlled at any point along its trajectory (arc lengths are variable), the task of computing an admissible trajectory in voxel v_i is independent of the task for voxel v_j, except through the puncture points.

Since admissibility can be computed voxel-wise, the problem is reduced to one of finding an admissible path within a given voxel v from a starting configuration c_{in} on the entrance face to a goal configuration c_{out} on the exit face. From Laumond's work [35], a legal trajectory for a mobile robot with a minimum turning radius exists if and only if a trajectory consisting of some number of arcs of minimum turning radius and tangent line segments between these arcs exists. The search for such a canonical trajectory is performed over the centers of curvature for these arcs for a given path topology. Since a voxel is convex and since we've restricted the maximum size of the x and y dimensions of the voxel to be less than the minimum turning radius, it easily follows from Laumond's work that a trajectory exists between c_{in} and c_{out} if and only if a trajectory of the form (a_1, s, a_2) exists, where a_1 and a_2 are arcs along a circle with radius equal to the minimum turning radius, and s is the tangent line segment between them. The intuition behind the proof is that the maximum size of the voxel reduces the space of centers of curvature for a third arc to zero. By applying Laumond's result to a voxel at a time, trajectory admissibility can be shown for a robot of any shape in any environment, rather than just a circular robot in a polygonal world.

Since the selection of the polarity (left or right turn) of the arcs completely determines the tangent segment between them, we need to check only four possible trajectories:

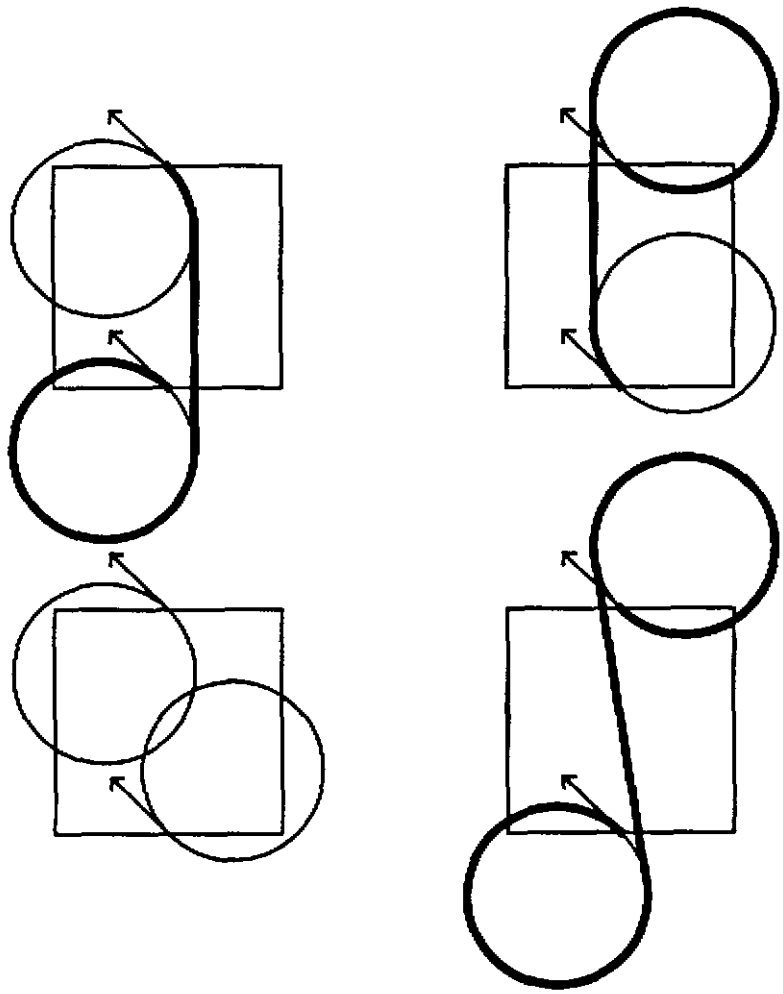


Figure 28: Testing Admissibility of Trajectories

(left, straight, left)
(left, straight, right)
(right, straight, left)
(right, straight, right)

to determine whether or not an admissible trajectory exists. A trajectory is labelled as inadmissible if part of it falls outside of the bounds of the voxel, or if the distance between the centers of curvature of arcs a_1 and a_2 is less than twice the minimum turning radius, for arcs of opposite polarity. Violation of the latter condition leads to an inadmissible trajectory because no line segment s can exist between the two arcs. Figure 28 illustrates two configuration points for which no admissible trajectory exists. Of the four possibilities, the first three fail because the trajectory is in part outside of the voxel. The fourth fails because no line segment between the arcs can exist.

The constraint equations for testing the admissibility of a trajectory between two configuration points follows. Let (x_1, y_1, θ_1) and (x_2, y_2, θ_2) be the two configuration points. Let r_{min} be the minimum turning radius of the robot. Let r_1 be the turning radius of the first arc, such that $r_1 = r_{min}$ if the first arc is a right turn, and $r_1 = -r_{min}$ if it's a left. The radius of the second arc, r_2, is defined similarly. The centers of the two arcs are defined by the following equations:

$$x_{c1} = r_1 \cos(\theta_1) + x_1 \tag{30}$$

$$y_{c1} = r_1 \sin(\theta_1) + y_1$$

$$x_{c2} = r_2 \cos(\theta_2) + x_2$$

$$y_{c2} = r_2 \sin(\theta_2) + y_2$$

If the trajectory is an S-turn (i.e., $r_1 = -r_2$), then the following constraint must be satisfied for the trajectory to be admissible:

$$(x_{c1} - x_{c2})^2 + (y_{c1} - y_{c2})^2 \geq (2 r_{min})^2 \tag{31}$$

In order to test whether or not the trajectory lies entirely within the voxel, we define a few other parameters. Let θ_{r1} and θ_{r2} be the angles measured from the positive y-axis to the radial line segment connecting the centers of the two circles as measured in coordinate systems with origins at the centers of the first and second circle respectively. Let θ_{s1} and θ_{s2} be the angles at which the trajectory leaves the first circle and arrives at the second circle respectively. We have the following equations:

$$\theta_{r1} = arctan\left(\frac{y_{c2} - y_{c1}}{x_{c2} - x_{c1}}\right) - \frac{\pi}{2} \tag{32}$$

$$\theta_{r2} = \theta_{r1} - \pi$$

$$d_r = \frac{\sqrt{(x_{c2} - x_{c1})^2 + (y_{c2} - y_{c1}2)}}{2}$$

$$\theta_e = \frac{\pi}{2}$$

$$\theta_o = arccos(\frac{r_{min}}{d_r})$$

$$\theta_{s1} = \theta_{r1} + sign(r_1)\theta_{e\ or\ o}$$

$$\theta_{s2} = \theta_{r2} - sign(r_2)\theta_{e\ or\ o}$$

where d_r is half the distance between the centers of the circles and θ_e and θ_o are the angles between θ_s and θ_r for turns of even (e.g., left-left) and odd (e.g., left-right) polarity respectively. The equations for determining the values of x and y along the arcs as a function of θ are:

$$x_{b1} = r_1(\cos(\theta_1) - \cos(\theta)) + x_1 \tag{33}$$

$$y_{b1} = r_1(\sin(\theta_1) - \sin(\theta)) + y_1$$

$$x_{b2} = r_2(\cos(\theta_2) - \cos(\theta)) + x_2$$

$$y_{b2} = r_2(\sin(\theta_2) - \sin(\theta)) + y_2$$

The bounds on x and y as the trajectory follows an arc are computed by using the SUP-INF method on the above equations. For left turns, θ ranges from θ_1 to θ_{s1} for the first circle and from θ_{s2} to θ_2 for the second circle. For right turns, θ ranges from θ_{s1} to θ_1 for the first circle and from θ_2 to θ_{s2} for the second circle. The bounds on these equations can be compared against the x and y voxel bounds. The equations for θ_{s1} and θ_{s2} can be checked similarly for violations of the θ voxel bounds.

Testing the Admissibility of a Voxel Sequence

A simple technique for checking the admissibility of a voxel sequence $(v_1, v_2, ..., v_N)$ is to discretize the connecting faces into puncture points (such that there are d_v points along each dimension) and then use a dynamic programming approach to propagate admissible trajectories through the sequence. Consider the i-th voxel in a sequence. Let f_{in} and f_{out} be the entrance and exit faces respectively of the voxel v_i. Faces f_{in} and f_{out} are both subspaces of dimension two. Assume we have found the subspace of f_{in} (called f_{in}^*) that defines the termination configuration points of all admissible paths from voxel v_1 to the exit

face of a voxel v_{i-1}. Define f^*_{out} to be the set of *all* configuration points c_{out} in f_{out} such that an admissible sub-trajectory exists from at least *one* configuration point c_{in} in f^*_{in} to the configuration point c_{out}. The subspace f^*_{out} is then the set of all termination configuration points of all admissible paths from voxel v_1 to the exit face of voxel v_i.

The search for an admissible trajectory proceeds as follows: the subspaces f^*_i defining the set of termination configuration points for all admissible trajectories from voxels v_1 to v_i are constructed from $i = 1$ to $i = N$. By induction, iff f^*_N is empty, then there are no admissible paths through the sequence of voxels. Furthermore, if f^*_i is determined to be empty for any $i < N$, the search can terminate at i, since no legal paths can exist beyond this face.

If all of the configuration points on the entrance face of the voxel can reach all points on the exit face, then the complexity of the above algorithm is $O(N d_v^4)$. This complexity is unacceptably high. Unfortunately, it is nearly always the case in large voxels where the robot has space to maneuver. Fortunately, we can take advantage of continuity in the kinematic equations of the robot. Typically, the points reachable on the exit face by a point on the entrance face are grouped together due to continuity in the kinematic equations. The following algorithm uses this property to reduce the complexity of the search.

Equations 30 through 33 in the previous section define the kinematic constraints on the NAVLAB. The input parameters are individual entrance and exit configuration points. We would like to develop an algorithm to evaluate the constraints between entrance and exit *faces*. For a given pair of faces, f_{in} and f_{out}, we can use Equations 30 to compute the centers of the left and right turning circles that are the maximum distance from the exit face. These turns define boundaries on the envelope of trajectories between the two faces. By allowing the configuration parameters to vary over the exit face, we can evaluate the S-turn constraint (Equation 31) and the boundary constraints (Equations 33) to classify the entrance-exit face pairs as:

1. *Admissible:* every point in f_{out} is reachable from at least one point in f_{in} via a legal trajectory.
2. *Inadmissible:* no points in f_{out} are reachable from any points in f_{in}.
3. *Both:* at least one point (but not all) in f_{out} is not reachable from any point in f_{in}.

We replace the cross-mapping portion of the above algorithm with the following. Instead of a set of individual configuration points, let f^*_{in} be a quadtree representing the configuration points in f_{in} reachable from the starting configuration. The algorithm constructs a quadtree f^*_{out} of configurations on the exit face reachable from f^*_{in}. For purposes of clarity, assume that each vertex in the quadtree has four sons. A son can be another vertex, an admissible leaf node, or an inadmissible leaf node. Admissible and inadmissible leaf nodes define reachable and unreachable configurations respectively. Let f_i and f_o be the calling parameters of the algorithm (nodes in the quadtrees). Initially, f_i is set to f^*_{in} and f_o is set to f_{out},

a leaf node representing the entire exit face. Upon termination, f_o points to the root of the quadtree f^*_{out}. Let MAXRES be the maximum resolution allowed for subdividing faces.

1. If f_o is an admissible leaf node, return.
2. If f_i is an inadmissible leaf node, return.
3. If f_i is a vertex, let $f_{i,r}$ be a son of f_i. Recur at Step 1 with $(f_{i,r}, f_o)$, for each son. Return.
4. The node f_o is either a vertex or an inadmissible or untested leaf node. Test the admissibility between f_i and f_o.
5. If admissible, the node f_o must be modified. If f_o is a leaf, relabel it as admissible. Otherwise, delete its subtree and replace it with an admissible leaf. Return.
6. If inadmissible, return.
7. The classification is both. If the resolution of f_o is MAXRES, label it as inadmissible and return.
8. If f_o is a leaf, subdivide it into four subfaces and replace it with a vertex pointing to the four sons.
9. Let $f_{o,r}$ be a son of f_o. Recur at Step 1 with $(f_i, f_{o,r})$, for each son. Return.

The power of the above algorithm is that groups of admissible trajectories can be propagated from the entrance to the exit face of a voxel in a single operation. Figure 29 illustrates the mapping between a subface on the entrance face and a quadtree on the exit face of a voxel. The dimensions of the voxel are 6.4 meters by 6.4 meters by 40 degrees (θ is the vertical axis). The minimum turning radius is 8 meters.

The complexity of the above algorithm is roughly equal to the number of subfaces in the entrance face quadtree multiplied by the number of subfaces in the exit face quadtree. Thus, the complexity of the kinematic constraint propagation algorithm is considerably lower than the dynamic programming approach for large voxels.

Goal Specifications

The focus of this section is on techniques for planning a sound trajectory from a starting point to a goal point within a configuration space. As described in the previous section, this task is part of the larger task of global navigation. The bounds on the local planning space may be determined by the field of view of the robot's sensors or by the scope of map information in the vicinity of the robot. The planning goals are set to meet global objectives. The "goal" of the robot at the local level may be to reduce the distance to some distant landmark (e.g., head in some general direction), to ensure coverage of all terrain with the sensors as the robot moves, or to get into position to see a landmark.

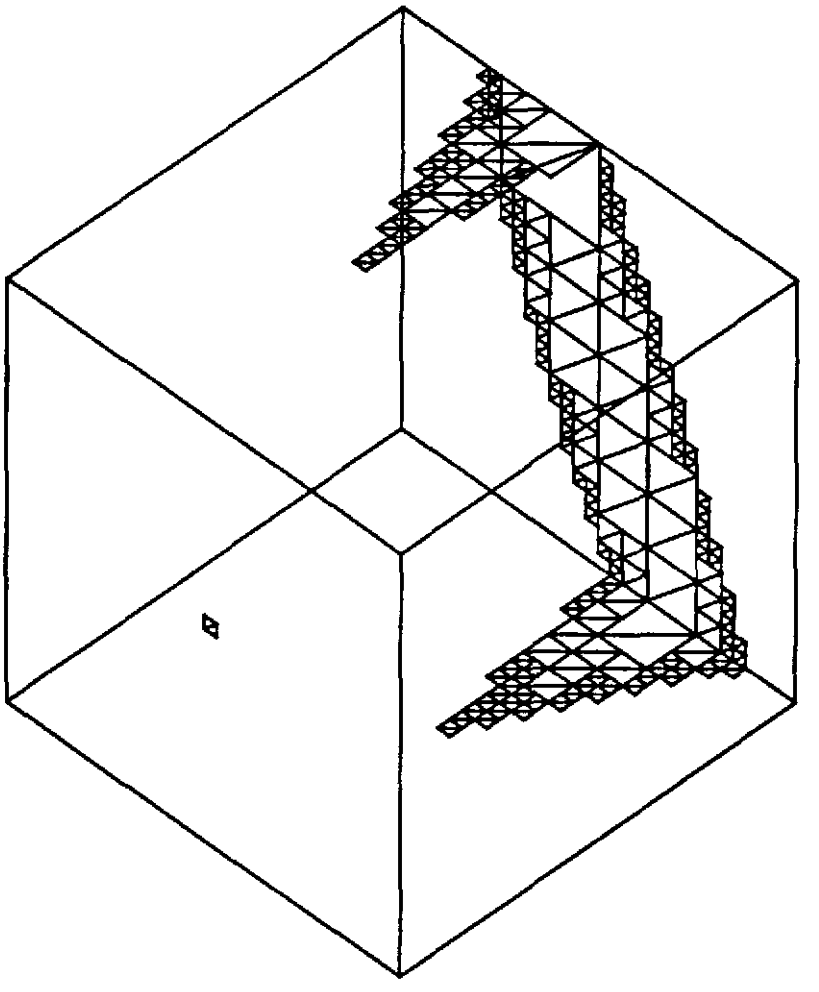


Figure 29: Quadtree Propagation of Admissible Trajectories

Determining Goal Configurations

If we model the field of view of the robot's sensor, the shape of landmarks, and the perimeter of the local environment as polygons, then the problem of determining coverage and the visibility of landmarks can be cast as a polygon intersection problem. The best goal configuration for determining coverage is one that maximizes the size of the new area seen while ensuring that this extension is connected to the existing area. Assuming that the viewframe and local environment polygons are convex, we can compute the new area seen by the sensor by subtracting the area of intersection of the viewframe with the local environment polygons from the area of the viewframe as shown in Figure 30. The area of the intersection alone can be used as an evaluation function for landmark visibility. The larger the intersection, the more of the landmark that is visible.

In general, computing the intersection of two polygons is an expensive operation. In the case that the polygons are simple, the complexity of the algorithm is the product of the number of vertices of the two polygons [45]. For convex polygons the intersection algorithm is linear in the total number of vertices in both polygons [45], but the algorithm does not lend itself well to evaluation across a voxel of configuration parameters (as is needed in the next section). For the NAVLAB, we can make a few simplifying assumptions. First, the robot extends its local environment by placing the distal bound of its field of view as far ahead as possible while keeping the proximal bound interior to the local environment. Second, the

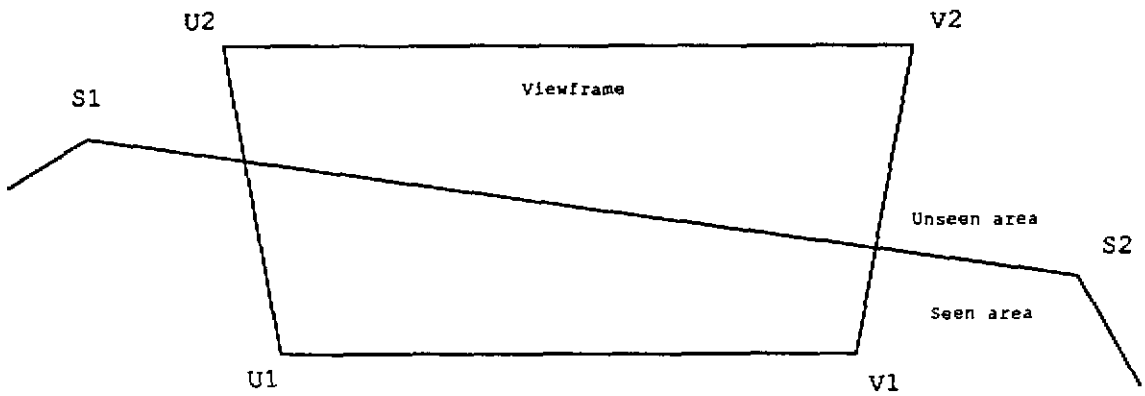


Figure 30: Measures for Evaluating Polygon Overlap

sequence of images forming the local environment can be approximated by a convex polygon with a small number of line segments. Thus, to ensure that the new area seen is maximized while overlap is maintained, we need only test that the two proximal vertices of the viewframe are both interior to and close to the bounds of the local environment.

Let u_1 and v_1 be the proximal vertices of the local environment (see Figure 30). Let s_1 and s_2 be the endpoints of a bounding line segment. Define the following vectors: $q_1 = s_2s_1$, $q_2 = s_2s_1$, and $q_3 = s_2v_1$. The two proximal vertices of the viewframe are interior to and within l_r of q_1 if the following conditions hold:

$$0 \leq \frac{q_1 \times q_2}{\|q_1\|} \leq l_r \tag{34}$$

$$0 \leq \frac{q_1 \times q_3}{\|q_1\|} \leq l_r$$

The goal condition holds for the entire environment if the two proximal vertices are interior to all bounds and within l_r of at least one bound.

A similar approximation can be derived for landmark visibility. We begin by bounding the landmark by a circle of radius r (see Figure 31). Define w_i to be the vector formed by subtracting the tail endpoint (as determined by a counterclockwise ordering) of the i-th vector of the viewframe from the center of the landmark circle. Define p_i to be the i-th vector of the viewframe. The entire landmark is visible in the

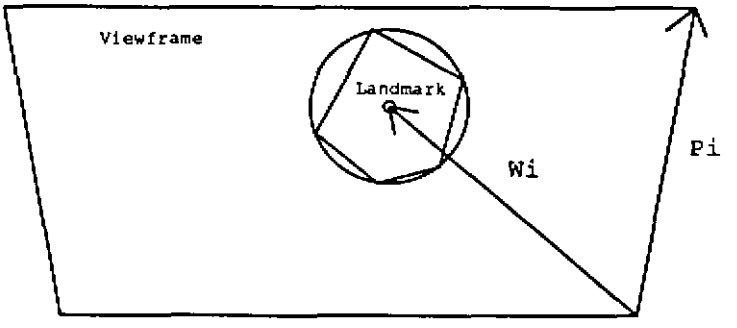


Figure 31: Determining Landmark Visibility

viewframe if the following condition holds for *all i* (vectors of the viewframe):

$$\frac{p_i \times w_i}{\|p_i\|} \geq r \tag{35}$$

Representing the Goal Space

As previously mentioned, typically more than one configuration point satisfies the goal requirements. Thus, we have a goal *space*. In order to facilitate planning, we would like an efficient way of representing this space. An octree can be constructed representing this space in a fashion analogous to that for admissible (safe) configurations described previously. For a given voxel of configuration points, we can compute bounds on the positions of the four viewframe endpoints. These bounds can be used in conjunction with the SUP-INF method to compute upper and lower bounds on Equations 34 to 35. Thus, we can classify the voxel as one of the following:

- *Admissible:* all configurations in the voxel are goal configurations.
- *Inadmissible:* none are goal configurations.
- *Both:* the voxel possibly contains a mixture.

We can recursively construct an octree representing the goal configurations, subdividing voxels classified as *both* until we reach a maximum resolution.

Searching for the Best Trajectory

The previous sections have illustrated how to determine whether a sequence of voxels in the configuration contain an admissible (kinematically correct) trajectory and whether a voxel contains goal configuration points. With this capability, we can search the space of voxels to find a path to a goal point. In this section, we define heuristic search and illustrate how local path planning can be cast as a heuristic search problem. We then discuss the heuristics used to efficiently search the space.

Heuristic Search

The search technique we employ is heuristic search. We briefly describe the algorithm here; see [44] for a detailed explanation. The search space consists of a set of *states*, one of which is the *start* state and at least one of which is a *goal* state. The algorithm maintains a list of states x, called the *open list*, ordered according to increasing value of a heuristic *evaluation function*, $f(x)$. Initially, the open list consists only of the start state. On each iteration of the search, the algorithm selects the state on the open list with the lowest f-value and *expands* it. A state is expanded by computing the set of states directly reachable from it via a single *operation*. The f-values are computed for each state in this resultant set, and the set is added to the open list. If a state in the set already exists on the open list, its f-value is replaced with the new value if lower. Backpointers are maintained from each state in the set to the original state, which is deleted from the open list. The algorithm terminates when a goal state is reached.

The heuristic evaluation function has the following form:

$$f(x) = g(x) + h(x) \tag{36}$$

where $g(x)$ is the cost of the path from the start state to state x and $h(x)$ is an *estimate* of the remaining cost from state x to the goal. When a state x is expanded to a state y, the value of $g(y)$ is computed by adding the cost of reaching y from x to $g(x)$. The estimate $h(y)$ is computed and is added to $g(y)$ to get $f(y)$. If the function $h(x)$ is a lower bound on the actual remaining cost to the goal for each state x, then the algorithm is guaranteed to find the optimal path to the goal.

In our implementation of heuristic search, the states are the subfaces on the voxel faces. The total number of states is determined by the number of voxels in the *union* of the admissibility octree and the goal configuration octree discussed above. Constructing both octrees before beginning the search is very expensive computationally. Furthermore, it is unnecessary since usually only a small part needs to be explored before the goal is found. Other planners have circumvented this problem by constructing the admissibility octree as it is searched [17, 18]. We extend this approach to the goal octree as well and show how the two can be constructed concurrently as the search proceeds. The rest of this section describes the expansion operation at the core of the heuristic search. The selection of an evaluation function is deferred to the next section.

Initially, the algorithm begins with a single face representing the start state on the boundary of a voxel enclosing the entire planning space (see Figure 32). Each time a state x is selected from the open list,

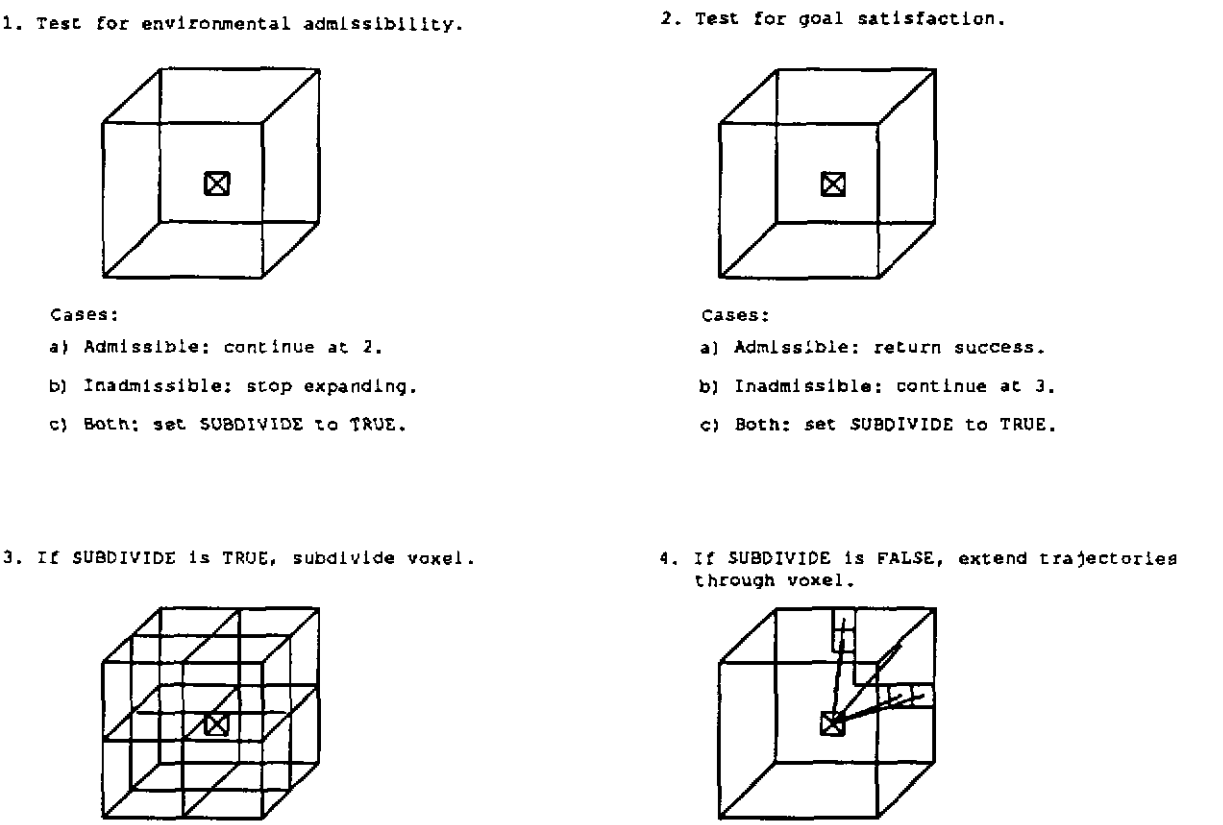


Figure 32: Expansion Operation for Heuristic Search

the following operations are performed on it. Let v be the voxel on which the subface x is attached. Initially, the flag SUBDIVIDE is set to FALSE.

1. Test v for admissibility. One of three cases results:
   a. *Admissible:* safe to move through the voxel. Continue the expansion at Step 2.
   b. *Inadmissible:* no room to maneuver through voxel. End the expansion and delete x from the open list.
   c. *Both:* mixture of safe and unsafe. Set SUBDIVIDE to TRUE.
2. Test v for goal satisfaction. One of three cases results:
   a. *Admissible:* all states in v are goal states. Stop and return success.
   b. *Inadmissible:* no goal states in this voxel. Continue with expansion at Step 3.
   c. *Both:* some goal states exist in the voxel. Set SUBDIVIDE to TRUE.

3. If SUBDIVIDE is TRUE, divide the voxel into eight subvoxels, v_i. Attach x to the appropriate voxel v_i, reenter it on the open list, and end the expansion. If x straddles four subvoxels after v is subdivided, divide x into four substates and reenter them on the open list.
4. If SUBDIVIDE is FALSE, compute the set of subfaces reachable from x by moving entirely within v. Attach backpointers from these subfaces to x and enter all of them on the open list. Delete x from the list.

If the maximum voxel resolution is reached in Step 1 the expansion is ended and x is deleted from the open list. If the maximum resolution is reach in Step 2, the expansion is continued at Step 3. Note that in Step , each subface of the quadtree is entered on the open list as a separate state, rather than together as a single state. This approach is adopted so that heuristics can be employed to selectively expand parts of the quadtree that are likely to lead to solution paths.

Selecting an Evaluation Function

The selection of the evaluation function affects both the optimality of the trajectories planned and the total search time required to find a trajectory. As discussed previously, *completeness* is more important than optimality (of distance travelled) for the local path planning problem. Therefore, we would rather have a suboptimal trajectory that is found in considerably less time than an optimal one.

The evaluation function consists of two parts: $g(x)$ and $h(x)$. We could choose $g(x)$ to be the length of the trajectory from the start state to the state x. With the appropriate $h(x)$, this selection would minimize the distance travelled. The shortest trajectories, however, are generally not the quickest to find. Testing voxels for admissibility is the most time consuming operation performed by the planner. Short trajectories require the robot to pass close to obstacles where voxels are subdivided to a high resolution to define the obstacle boundary. Instead, we choose $g(x)$ to be the total *number* of voxels through which the trajectory passes from the start state to x. Thus, trajectories that pass through a small number of voxels are favored. Since large voxels minimize the number of voxels per unit distance travelled, trajectories that avoid highly convoluted sequences of small voxels are explored first.

In choosing the function $h(x)$ there are two problems: determining the goal point to which to estimate the remaining cost and computing the estimate itself. In the case that the planner is attempting to minimize the distance to a goal point, the goal point is given. A good lower bound on the remaining cost is the straight-line distance (in voxel units) from the state x to the goal point. As shown in Figure 33, the voxel distance includes smaller voxels needed to reach the boundaries of the maximum-size voxels.

In the case that we have a goal *space*, the problem is more difficult. The true value of $h(x)$ is the minimum cost to *any* point in the space. The best estimate is achieved by constructing the entire goal octree before planning. For each state x evaluated, the goal octree could be searched to find the leaf with the lowest cost (straight-line distance) to x. This estimate is very expensive computationally. At the other extreme, we could use an estimate of $h(x) = 0$ for all x. Although this estimate is trivial to compute, it provides no information to guide the search, and the search becomes exhaustive.

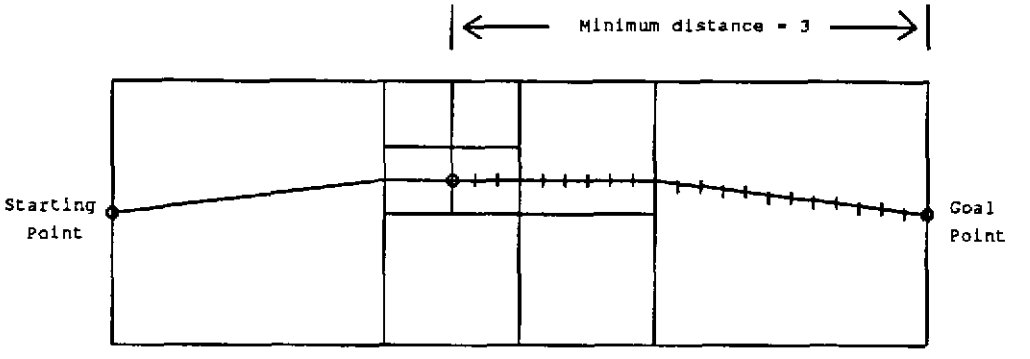


Figure 33: Estimate of Remaining Distance in Voxels

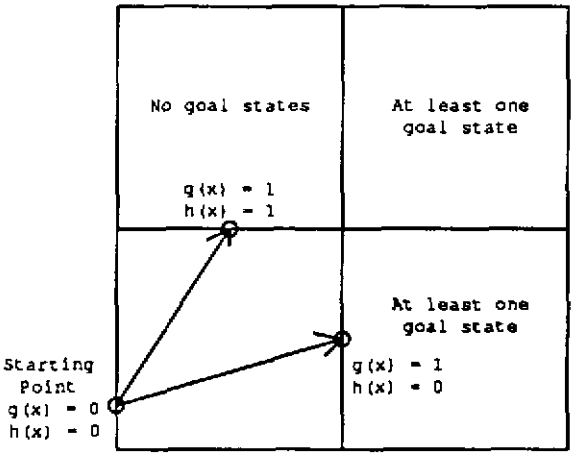


Figure 34: Estimating the Remaining Cost to the Goal

A good compromise is to construct enough of the goal tree as needed to guide the search. In the expansion algorithm of the previous section, the third step is to subdivide the voxel into eight subvoxels. We then test these subvoxels for goal satisfaction. If the subvoxel contains at least one goal state (classified as *admissible* or *both*), the subvoxel is assigned a *goal value* of zero. If the subvoxel contains no goal states, its goal value is set to the value of its lowest neighbor plus one. Thus, the goal value represents a lower bound on the remaining distance from the subvoxel to a voxel in the goal octree. The last step of the expansion algorithm is to determine the set of states, x_i, reachable from the state x. Each of these states borders voxel v on one side and a corresponding voxel v_i on the other. The value of $h(x_i)$ is set to the goal value of v_i. The process by which the remaining cost is estimated is illustrated in Figure

34. States that border voxels with at least one goal state have a remaining cost estimate of zero (since the goal state may lie infinitesimally close to the state), while the others have a remaining estimate of one (the minimum voxel distance to the goal octree).

Even with a heuristic evaluation function that minimizes the number of voxels through which the solution trajectory will pass, the search can be slow at times. The function can be tweaked to improve the run time. A standard approach is to increase the estimate of the remaining cost, $h(x)$, such that it is *greater* than the true remaining cost. This sacrifices optimality in exchange for reduced search time. With such a function, the planner favors branches of the search extended the farthest, thus the search takes on a depth-first flavor. This tweak is appropriate for typical NAVLAB scenarios, since most local planning does not involve convoluted paths, but minor deflections to straight-line paths, where a depth-first strategy yields faster results. Another tweak is to heavily weight (via a large $h(x)$ value) states that do not explore new voxels. This tweak prevents the planner from searching within existing voxel sequences to find better paths until absolutely necessary.

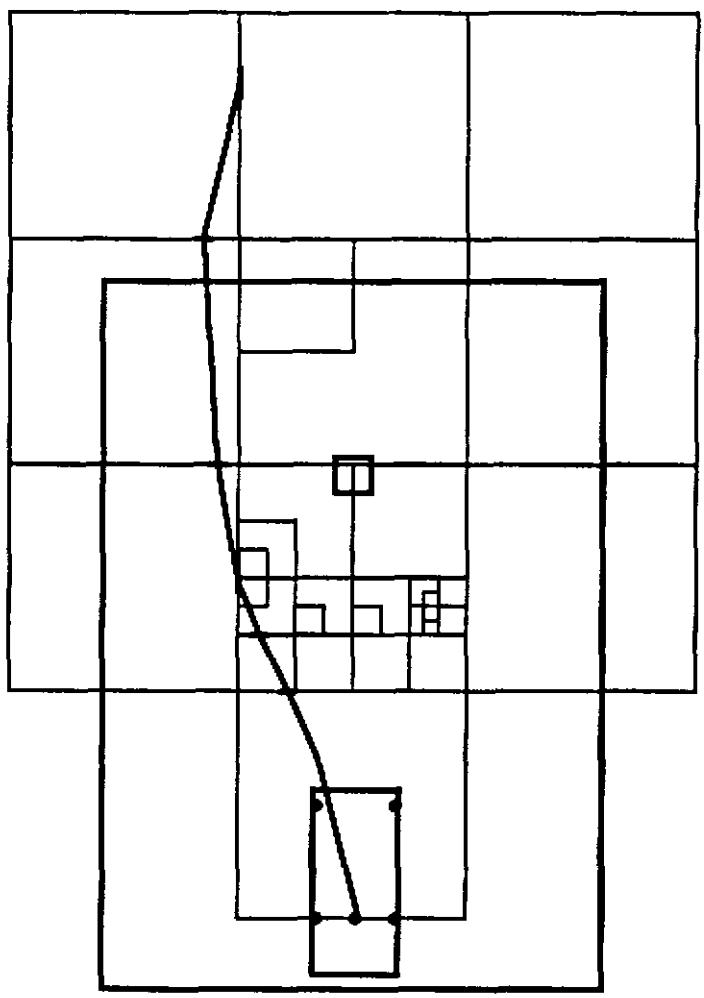


Figure 35: Example of Planned Path Around an Obstacle

Figure 35 illustrates a planned path for the NAVLAB. In this figure, a trajectory is found around an obstacle. The large boldface rectangle is a flat surface bounding the planning space, and the boldface square is the obstacle. The vehicle and solution trajectory are shown in boldface, and the remaining

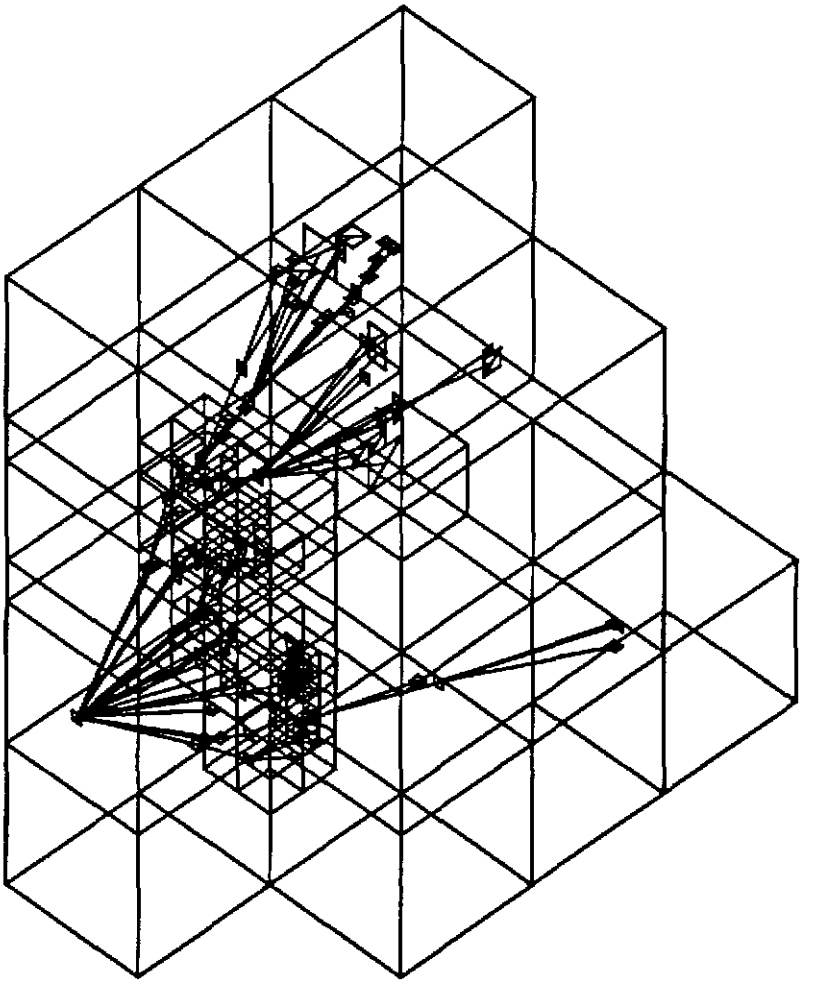


Figure 36: Trajectories searched for Previous Example

squares illustrate the xy projection of the expanded octree. In this example, the remaining cost estimate is doubled, and states not yielding a new voxel expansion are given an added weight of 10. The number of voxels expanded to find the solution was 182. An off-axis projection of the expanded octree is shown in Figure 36. In this figure, all search paths are shown along with the subfaces between voxels (shown as squares with one diagonal). Note that the planner is able to search quickly through the first large voxel then must subdivide the second large voxel to find the boundary of the obstacle before moving through the third. Finally, Figure 37 illustrates an off-axis projection of the solution path.

Path Smoothing

It is desirable for a mobile robot to operate smoothly [50]. The effects from abrupt changes in the robot's control can range from excessive wear on the robot's parts to instability during execution. Smooth control of the robot often means slowly changing control parameters. Unfortunately, if we attempt to smooth as we plan, the dimensionality required to represent separate, smooth paths uniquely is prohibitively high. We therefore treat smoothing as a posteriori process that runs after we have planned a coarse path. This process amounts to selecting the smoothest path that passes through a connected sequence of voxels.

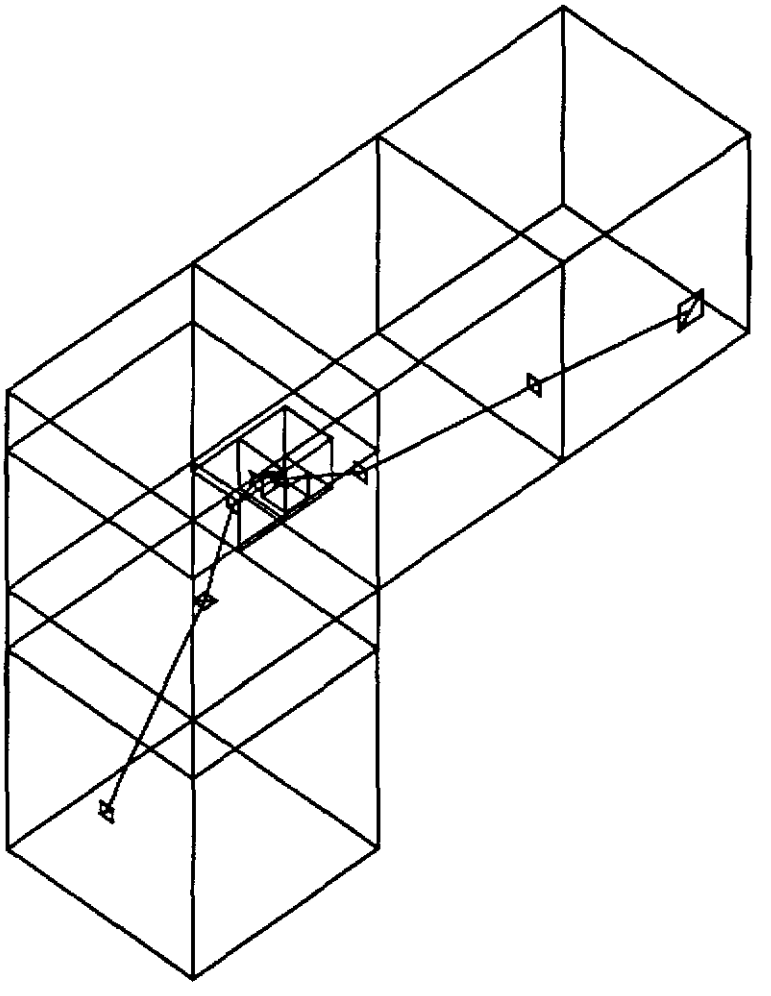


Figure 37: Solution Trajectory for Previous Example

Finding the Smoothest Trajectory

The end result of local path planning is a sequence of states (subfaces) extending from the start state to a goal state while residing entirely within a connected sequence of N voxels. An N-tuple of configuration points $(c_1, c_2, ..., c_N)$ on the faces between adjacent voxels specifies a particular trajectory within this sequence. Without a loss of generality, we define "smoothness" as a scalar function of these configuration points $f(c_1, c_2, ..., c_N)$, such that smoothness is maximized when f is minimized. The problem then is to find the set of admissible configuration points c_i for a given voxel sequence, such that f is minimized. Path relaxation [50] was used to find smooth paths in the FIDO system. The technique starts with some initial trajectory, i.e. a set of configuration points c_i, and then adjusts each configuration point one at a time in the direction that minimizes f. The advantage of such a scheme is that it is fast, i.e. linear in the number of configuration points N. In order to guarantee that path relaxation and other gradient-descent schemes will find the set of configuration points c_i that minimizes f, the following two conditions must hold:

1. The function f must have no local minima over the domain of admissible c_i.
2. The space of admissible configuration points must have no points P in $(c_1, c_2, ..., c_N)$, such that all neighbors of P which have lower f values fall outside of the space.

Both conditions ensure that no points P will be reached such that f cannot be decreased by stepping to a

neighbor of P. Convex spaces of P satisfy the second condition, although some nonconvex spaces work as well.

The problem with a direct application of path relaxation as developed in FIDO [50], is that kinematic constraints can render the space P nonconvex. We extend path relaxation in this section to ensure that during relaxation the trajectories remain in the space P as f is minimized. Given a sequence of subfaces through a voxel sequence, we can select an arbitrary "starting" trajectory given by an N-tuple of configuration points. At each step in the relaxation process, we select some configuration point c_i to adjust in order to minimize f. As described above, for each c_i on the entrance face of a given voxel v, we precompute the set f_{out} of configuration points reachable on the exit face. Likewise, for each point c_i on the exit face, we compute the set f_{in} of configuration points that can reach c_i from the entrance face. Therefore, in order to compute the set of points in which we can locally adjust c_i, we take the *intersection* of f_{out} for c_{i-1} with f_{in} for c_{i+1}. Selecting a point from this set guarantees that the trajectory remains admissible.

It should be noted that some smoothness functions or spaces may not meet the above conditions for relaxation. If not, the space of trajectories must be searched using a global technique. If the function f is a cost function, that is, it is the sum of cost values between adjacent trajectory configuration points, then a dynamic programming approach [1] can be employed to find the set of configuration points that minimizes f, even if both conditions are violated. (The cost function requirement is sufficient although not necessary.) The disadvantage of dynamic programming is that the complexity is $O(NM^2)$, where N is the number of trajectory configuration points and M is the number of discrete values for each configuration point.

Smoothness for the NAVLAB

Since the NAVLAB is controlled by setting arc curvatures and lengths, we define a smooth path as one that minimizes the change in curvature from arc to arc. For a given pair of configuration points (c_{in}, c_{out}) on the entrance and exit faces respectively of a voxel v_i in an admissible sequence, there exists at least one admissible path that lies entirely within the voxel. For purposes of the proof, this path consists of an arc-segment-arc triplet (a_{in}, s, a_{out}), such that the arcs determine a circle of minimum turning radius. Turns of minimum turning radius, however, lead to very rough control. At points where the robot changes turning polarity, the curvature of the turning arc jumps from one extreme to the other instantaneously.

Smoother paths may exist between the configuration points that lie entirely within the voxel. We define a measure of the "smoothness" of a trajectory as:

$$C_S = \sum_{i=2}^{N} |k_i^{in} - k_{i-1}^{out}| + (1 - m_i)|k_i^{in} - k_i^{out}| + m_i(|k_i^{in}| + |k_i^{out}|) \tag{37}$$

where N is the number of arcs or line segments in the trajectory, k_i^{in} and k_i^{out} are the curvatures of the first

and second arcs respectively in the i-th voxel, m_i is one if a tangent segment exists between the two turns and zero if one does not, and k_0^{out} is the curvature of the last arc of the previous trajectory. We can maximize smoothness by selecting the trajectory with the minimum C_S. For large voxels, however, the "smoothest" trajectory may be grossly suboptimal in length. Define C_L to be the length of a trajectory. We select the path that minimizes:

$$C = W_S C_S + W_L C_L \tag{38}$$

where W_S and W_L are weights chosen to balance the desired level of smoothness with trajectory length.

A configuration point c_i on the entrance face of voxel v_i can be adjusted in four dimensions, two in the entrance face and two in the curvature space. Our smoothing process has two stages. First, we adjust the two entrance face dimensions until a local minimum is found. Second, we adjust the two curvature parameters until a minimum is found. The first step allows the trajectory to rapidly shrink in length with minimal interference from the two-radius constraint (since the initial trajectory consists of turns of minimum turning radius--the least restrictive case). The second step achieves the proper balance between trajectory smoothness and length by varying the curvatures.

Each half of the smoothing process operates as follows. The N configuration points are adjusted repeatedly in sequence until a maximum number of iterations has been reached or until no configuration point can be adjusted to reduce C (i.e., a local minimum has been reached). For each configuration point, c_i, we add and subtract a small increment to its two free parameters and evaluate C at the resultant four points. The point that reduces C by the greatest amount is chosen, provided the resultant trajectory neither violates the two-radius constraint nor leaves the bounds of the voxel sequence. If none of the four points are legal during the trajectory-shrinking stage, the polarity of the curvatures at c_i are reversed in an attempt to create legal points. If that attempt fails or if none of the legal points reduce C, then c_i is left unadjusted.

The smoothing process is illustrated in Figures 38 and 39. Figure 38 illustrates a voxel sequence for the NAVLAB before smoothing. The largest voxel is 6.4 meters by 6.4 meters by 40 degrees. The minimum turning radius is 8 meters. The weights W_S and W_L were set to 10.0 and 1.0 respectively. The trajectory found by the path planner is shown. It consists of turns of minimum turning radius connected by straight line segments. The crosses mark the points where arcs meet straight line segments or other arcs. Some voxels appear to contain more than two crosses; however, the "extra" crosses are due to a second voxel superimposed on the first (the sequence is three-dimensional). Before smoothing, the value of C was 55.39. After 10 iterations of smoothing in x, y, and θ with increments of 0.2 meters, 0.2 meters, and 1.25 degrees respectively, the value of C dropped to 54.28. After 10 iterations of smoothing in curvature (both incoming and outgoing arcs at each point) with an increment of 4 meters^{-1}, the value of C dropped to 46.40. The smoothed trajectory is shown in Figure 39. Note that the path is both shorter and smoother.

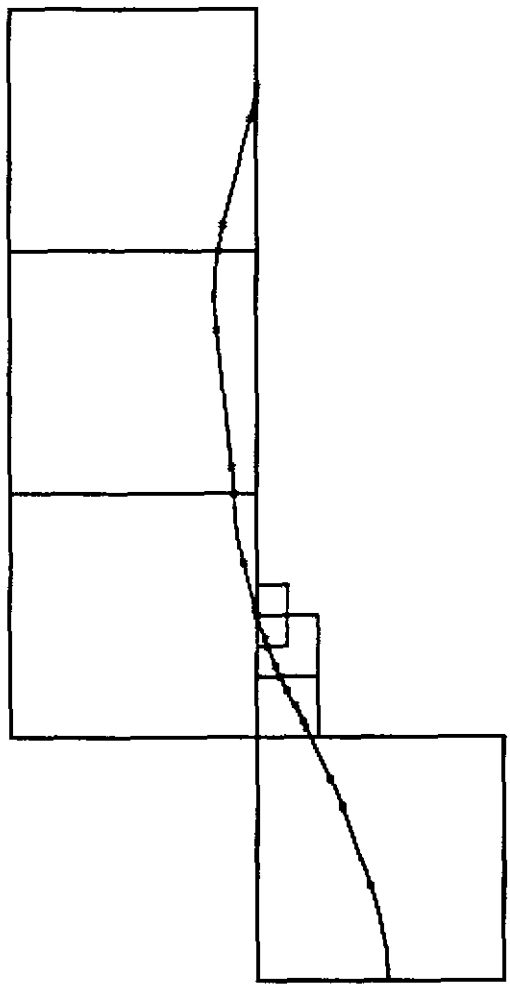


Figure 38: Voxel Sequence and Trajectory Before Smoothing

Experiments and Results

The environment modeling system and local path planner described in this section were implemented and tested on real and simulated data. The model functioned as the search space for the planning system. A typical environment is shown in Figure 40. In this configuration, the environment consists of three terrain meshes spanning an area approximately 12 meters wide and 12 meters deep. The meshes are shown here from an off-vertical axis to illustrate the height of the polygons in the meshes. In this example, the environment roughly consists of a flat center area, flanked by obstacles of various sizes on either side.

Figure 41 lists the processing results for eight typical terrain environments (sequences of three images) taken from a park adjacent to campus. For each sequence, the admissibility of a configuration space 12.8 meters by 12.8 meters by 40 degrees was computed by constructing the octree. The lowest resolution voxels (level 0) in octree are 6.4 meters by 6.4 meters by 40 degrees. The highest resolution voxels (level 5) are 0.2 meters by 0.2 meters by 1.25 degrees. The terrain pyramid covers an area 20 meters by 20 meters with the highest resolution terrain nodes (level 6) having dimensions of 0.31 meters by 0.31 meters.

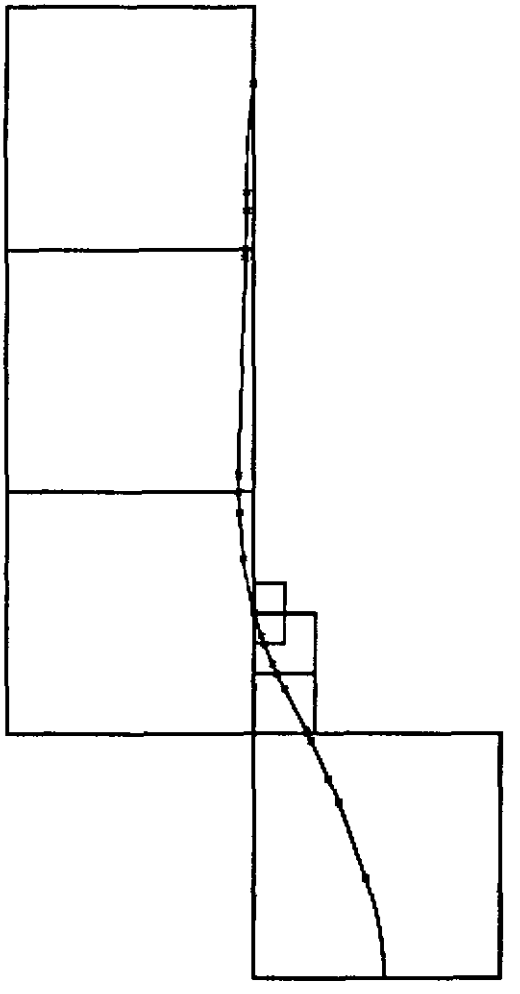


Figure 39: Voxel Sequence and Trajectory After Smoothing

The first row in the table lists the number of the figure that shows the terrain sequence. The next six rows indicate the number of voxels in the octree at each resolution. The eighth row lists the total number of states needed to represent the environment, and the ninth row lists the total number of terrain accesses required to construct the octree. If the configuration space were tested for admissibility everywhere at the highest resolution, the total number of states would be $(12.8\times12.8\times40)/(0.2\times0.2\times1.25) = 131072$. At the highest terrain resolution, the number of terrain nodes spanned by the undercarriage of the vehicle is 128. Thus, the total number of terrain accesses needed would be $131072\times128 = 16777216$. The tenth row in the table lists the ratio of this number to the terrain accesses in row nine.

As illustrated in the table, the speedup in terrain accesses ranges from one to three orders of magnitude. The actual reduction in terrain accesses and number of states realized by the octree approach varies with the terrain itself. Completely flat terrain can be represented with a single voxel, while pathological, undulating terrain can require many high resolution voxels, thus resulting in very little reduction. In order to make the numbers in the table more meaningful, an x,y slice through the configuration space at $\theta = 0$ (vehicle pointed forward) is superimposed on the meshes in Figures 42 through 49. Figures 42, 43, and 47 show terrain that is approximately flat or gently rolling. Figure 44

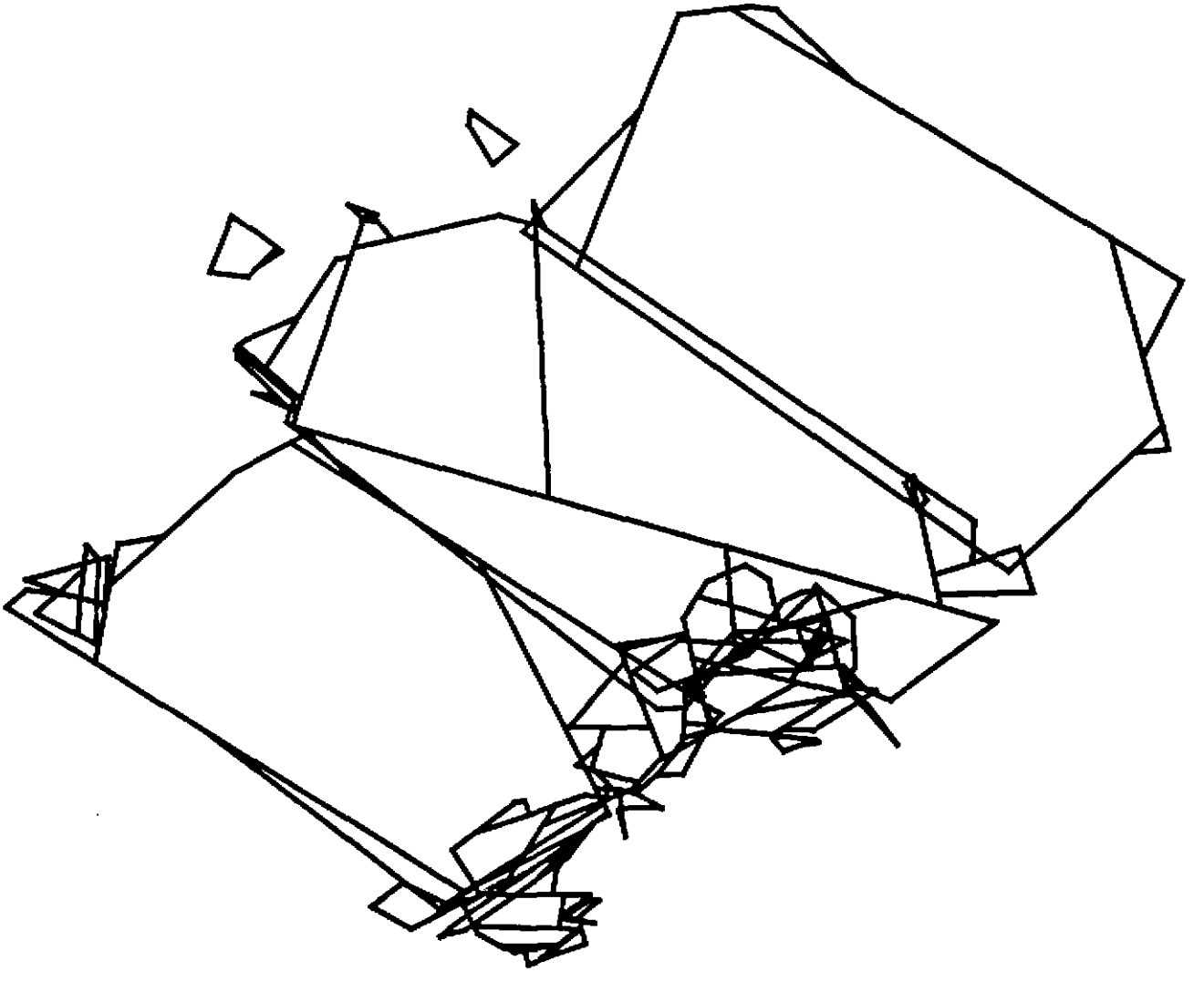


Figure 40: Typical Local Environment for NAVLAB

shows ditches to the left and a long trench to the right of center. Figures 45 and 46 show trees on one or both sides of the environment, and Figure 48 shows a tree directly in front of the vehicle. Finally, Figure 49 shows a tree on the left, and rocky terrain on the right.

A number of heuristics have been employed to expedite the construction of the octree. The most powerful of which is the observation that if the maximum difference in elevation across the terrain overlapped by the vehicle's body for a given voxel is less than the undercarriage height, then it is not possible for the body to intersect the terrain and the voxel can be labelled as admissible (with regard to body collision). This test is much faster than computing the positions of the four wheels, the tilt parameters of the undercarriage, and the intersection of the plane with the terrain itself. Note, however, that many cases arise (e.g., a steep slope) where the elevation disparity is greater than the undercarriage height, but no body collision occurs. In such cases, the system resorts to the brute-force method of evaluating admissibility.

Figure	42	43	44	45	46	47	48	49
Res 0	4	4	4	4	4	4	4	4
Res 1	32	16	32	30	32	20	40	32
Res 2	24	40	264	200	194	88	200	256
Res 3	0	88	1240	600	1264	104	680	1328
Res 4	0	136	2776	1500	4272	318	944	4360
Res 5	0	304	9688	5456	15984	1160	3352	13200
Total	60	588	14004	7790	21750	1694	5220	19180
Access	32520	64168	1276828	781705	1872169	171822	532955	1667601
Ratio	516	261	13.1	21.5	9.0	97.6	31.5	10.0

Figure 41: Octree Construction for Typical Terrain Environments

In addition to reducing the construction time, the data and constraints are transformed to reduce the total number of states needed to represent the admissibility. For example, in the case of sloped but approximately planar terrain, high-resolution voxels are needed to localize the positions of the wheels to determine that the plane of the undercarriage is roughly parallel to the plane of the terrain, and thus, that no collisions exist. By performing a coarse sampling of the image, the approximate slope of the terrain can be computed, and the terrain mesh can be rotated to an approximately level orientation. The tilt constraint must be corrected for offset orientation. The result is that the separation between the robot's body and the "adjusted" terrain becomes readily apparent in z, the elevation dimension, and large voxels can be cleared for admissibility in a single operation.

As discussed in the previous section, for reasons of robot responsiveness it is desirable to plan through more than one mesh. Care must be exercised in fusing these meshes together into a single environment. If the relative transforms between sensing poses of the robot is not known accurately, phantom "steps" can appear between meshes. This case can arise if the robot is travelling over a bump while the ERIM is digitizing an image. In such cases, the robot will perceive the misalignment between meshes as a step, possibly too large to climb. Currently, we use a crude algorithm for registering the elevation values of adjacent, overlapping images. This approach breaks down in rocky terrain. In the future we expect to use attitude sensors to correct each image.

The above environment model functioned as the search space for the local path planner. We illustrate a number of interesting features of the planner in isolated examples and evaluate it's performance on real terrain. As mentioned before, our evaluation function favors trajectories that require the fewest voxel expansions over trajectories that are short. This feature is illustrated in Figures 50 and 51. In both figures the boldface square obstacle is small enough to fit under the vehicle without colliding with the undercarriage but too large for a wheel to climb over it. In Figure 50, the obstacle is placed directly in

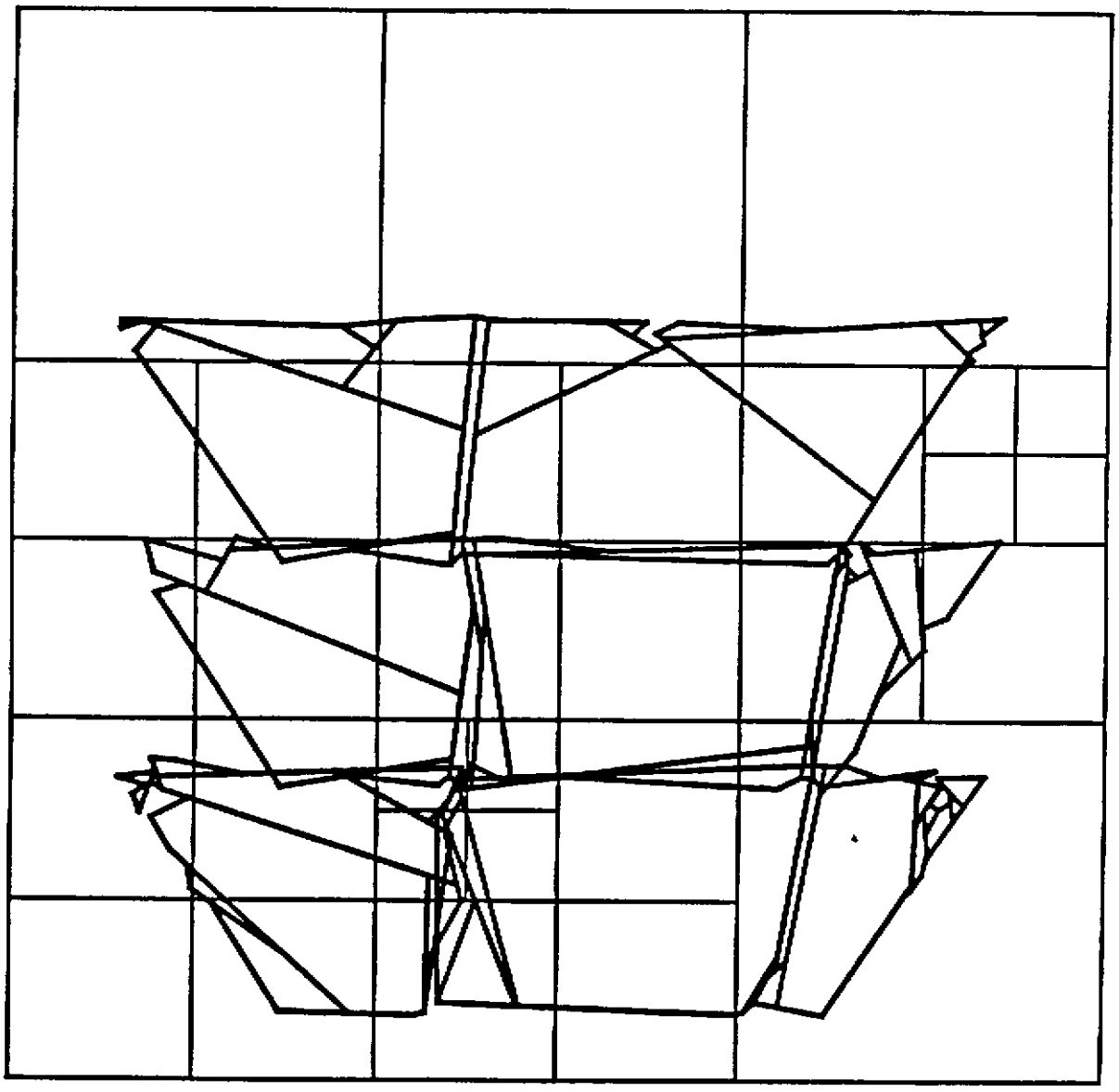


Figure 42: Admissibility of Flat Terrain

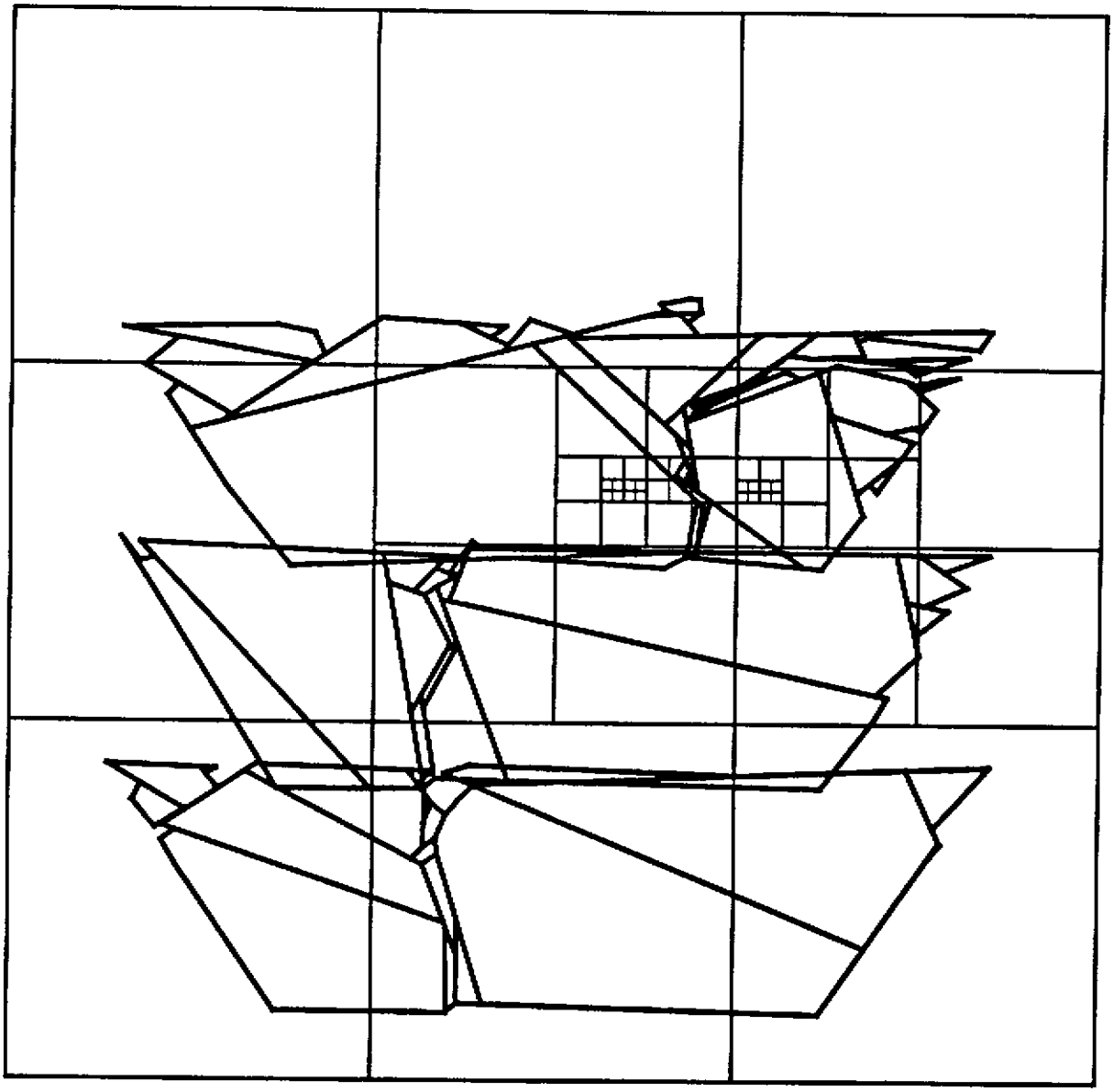


Figure 43: Admissibility of Flat Terrain

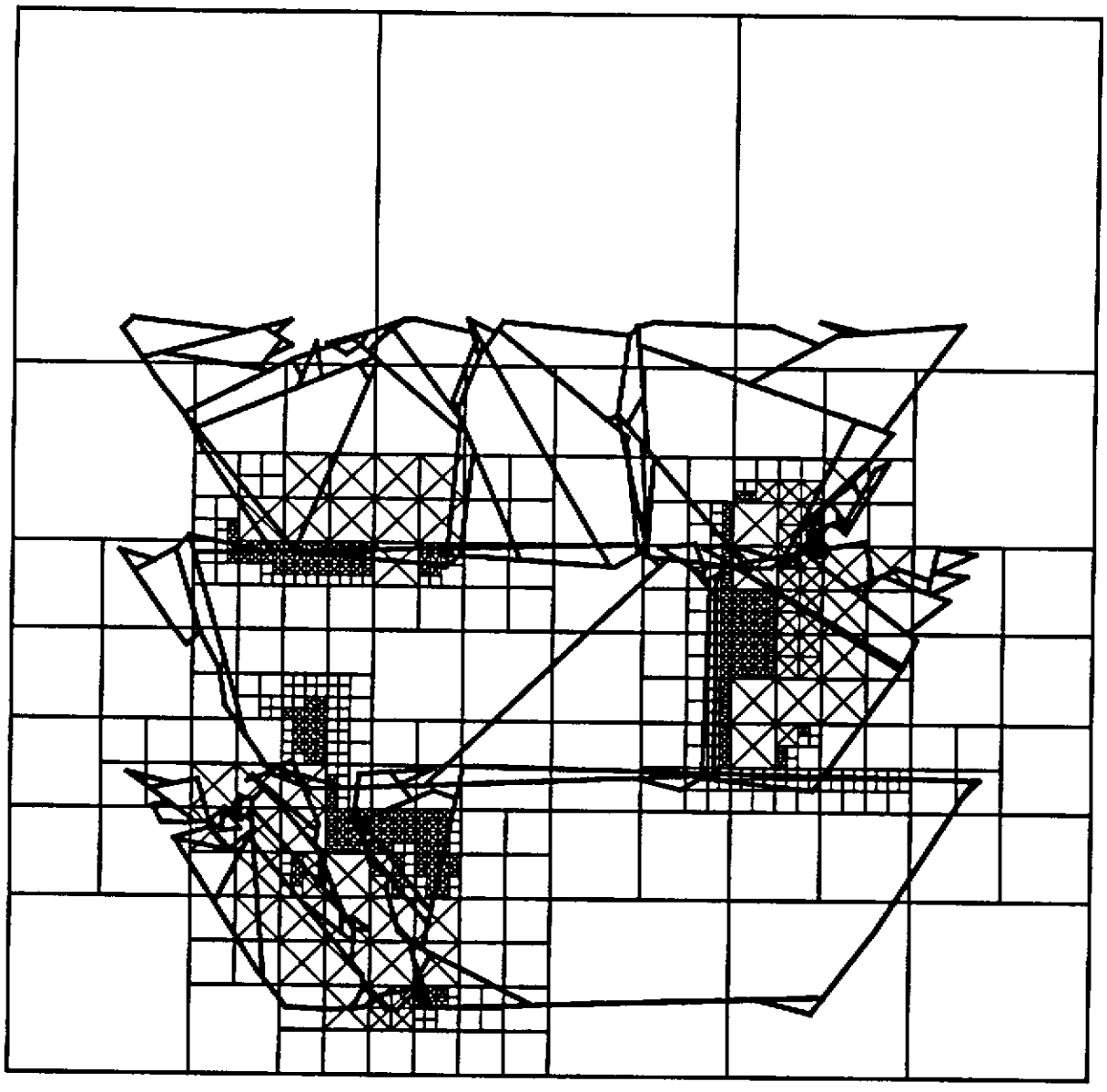


Figure 44: Admissibility of Ditch and Trench

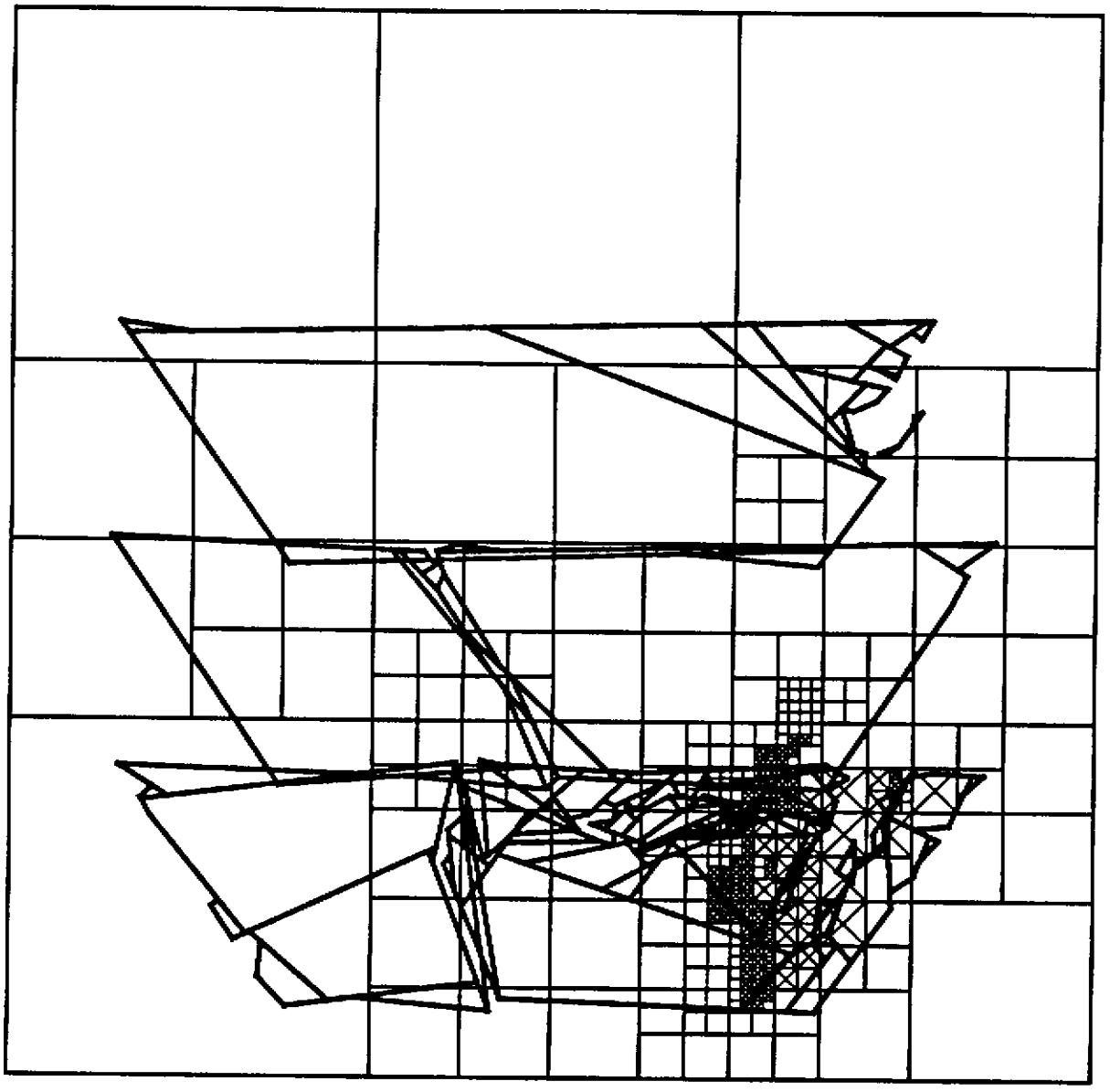


Figure 45: Admissibility of Tree Flanking Path

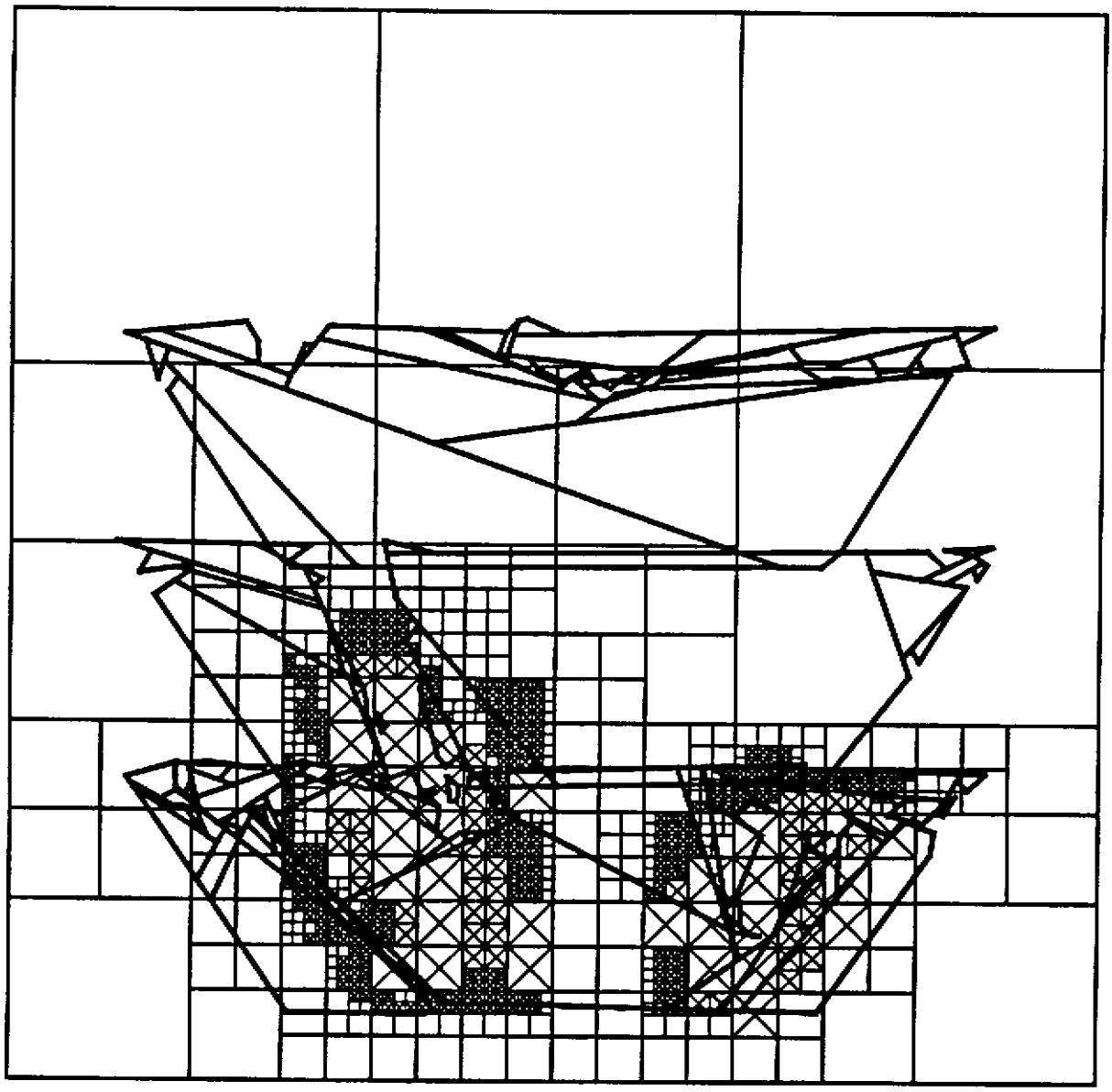


Figure 46: Admissibility of Tree Flanking Path

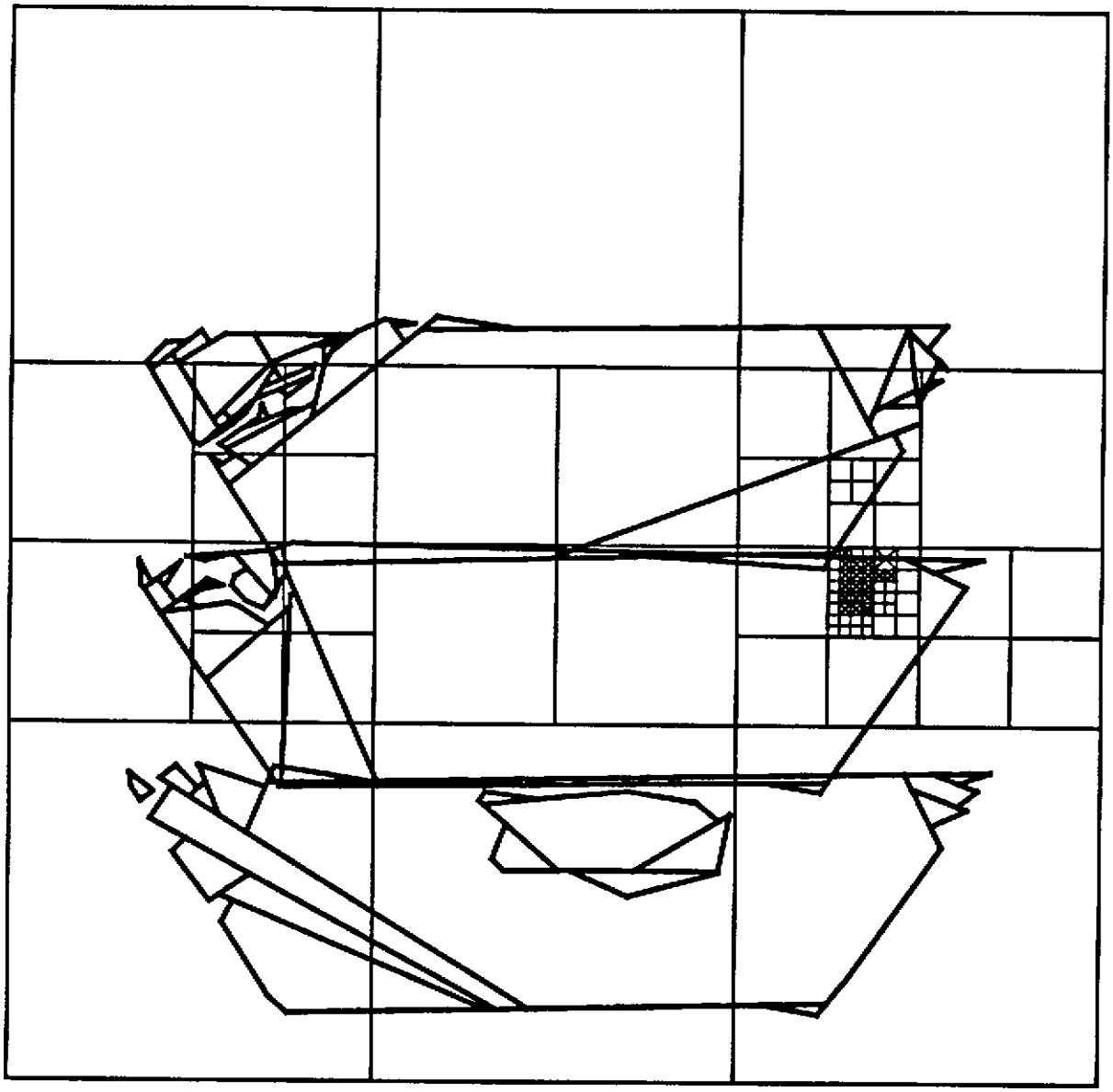


Figure 47: Admissibility of Flat Terrain

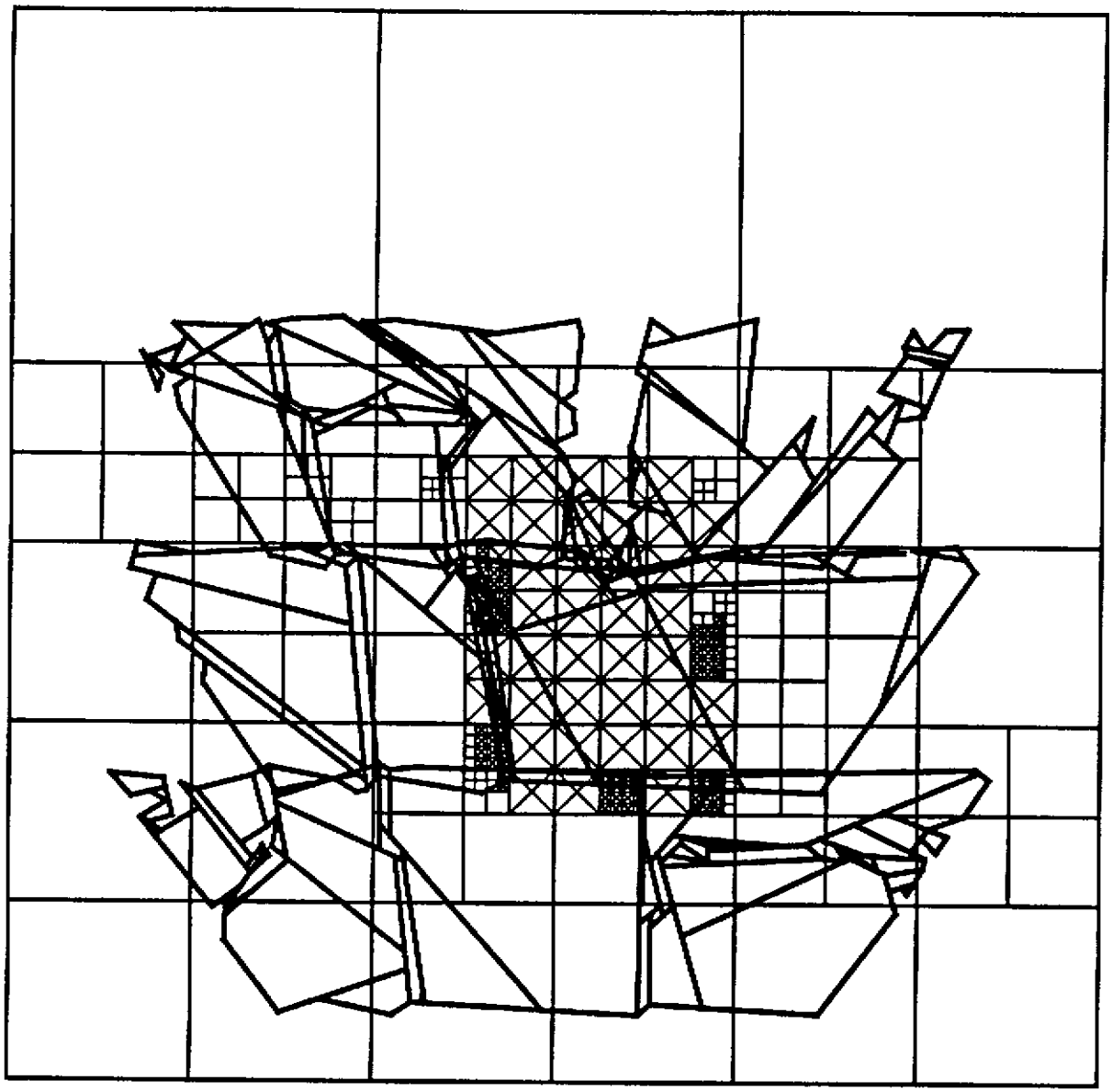


Figure 48: Admissibility of Tree in Front of Vehicle

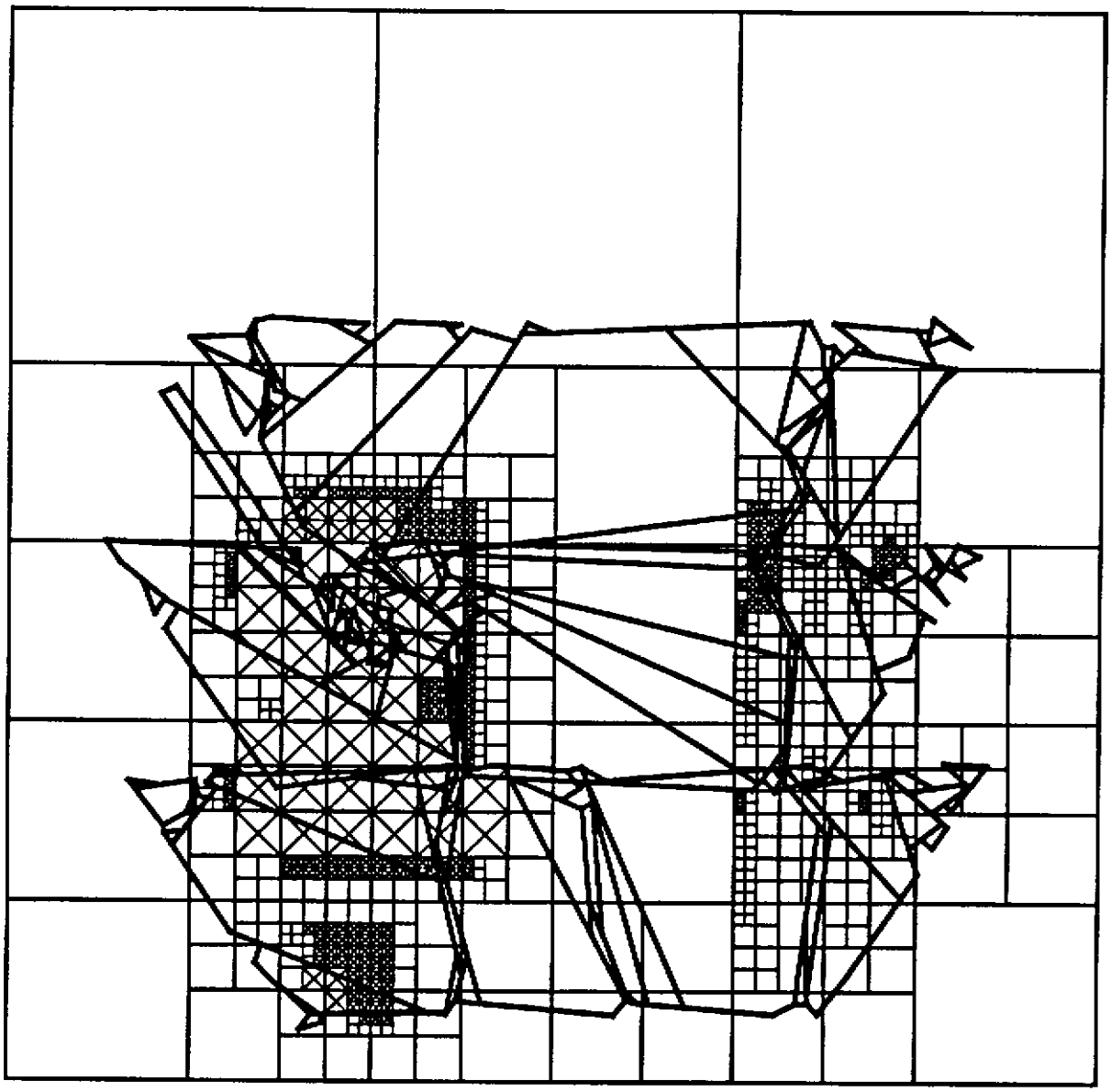


Figure 49: Admissibility of Tree and Rocky Terrain

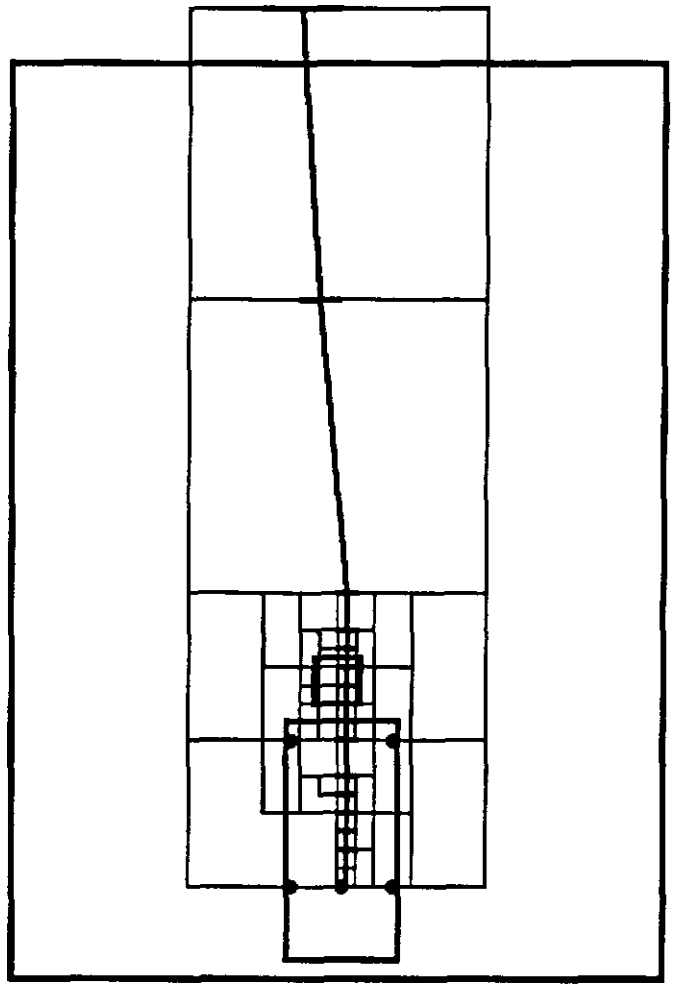


Figure 50: Searching for a Trajectory at a High Resolution

front of the vehicle. The planner does not have enough room to maneuver around the obstacle, so it is forced to find the solution trajectory that takes the vehicle over the object. A total of 230 voxels were expanded to find this trajectory. In Figure 51, the obstacle is moved far enough in front to permit the vehicle to maneuver around it. The planner finds a trajectory around the obstacle, rather than expending the computation needed to find the trajectory over the obstacle (by searching through small voxels). The total number of voxels expanded was reduced to 185.

As discussed above, it is important to know the approximate slope of the terrain through a coarse sampling of the data in order to "level" it before planning. Without this adjustment, the planner will still find a trajectory if it exists, but at a much greater search cost. In such cases, it must resort to high resolution voxels to ensure that the disparity in elevation values will not result in a body collision. This point is illustrated in Figures 52 and 53. In Figure 52, the "terrain" consists of a single horizontal polygon. The vehicle starts angled one degree to the left of center, and the goal is positioned directly in front of the vehicle just off the terrain. The solution trajectory and an xy slice of the expanded voxels are shown. A total of 3 voxels need to be expanded to find the solution. In Figure 53, the terrain slopes upward from left to right at an angle of 5 degrees. This tilt is not corrected before planning and the planner must expand 63 voxels to find the solution trajectory. The current system employs a crude tilt correction algorithm.

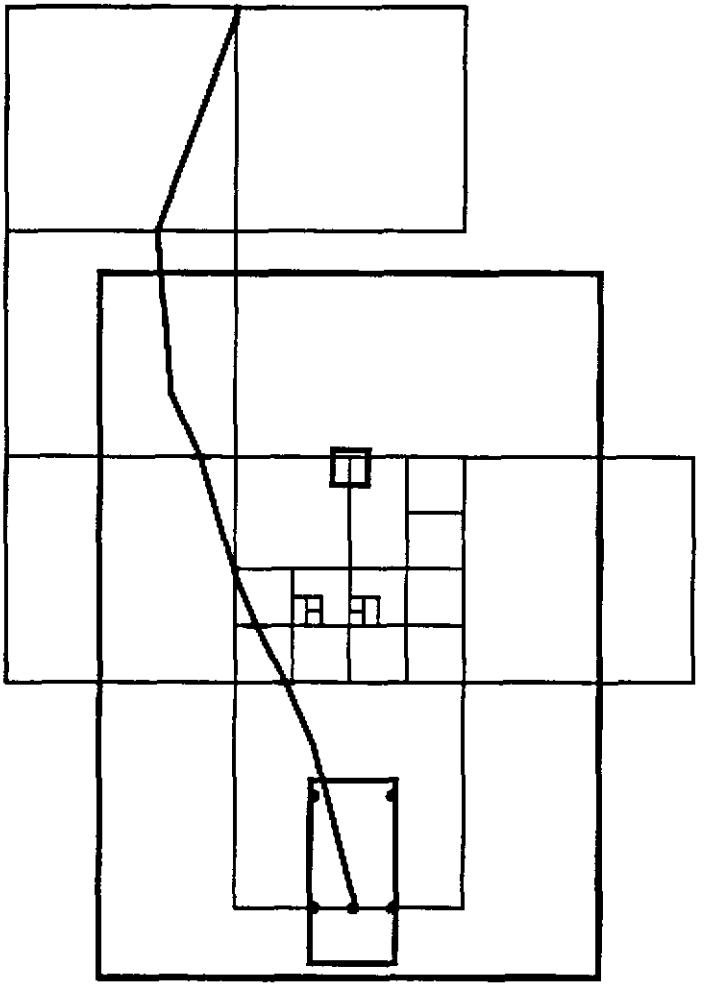


Figure 51: Searching Through Large Voxels to Reduce Computation

Another problem arises with the kinematic constraints in high resolution voxels. In such cases, even the highest resolution subfaces are large with respect to the voxel, such that no subface is reachable (in its entirety) via legal trajectories from another subface. Since the subfaces cannot be further subdivided, the end result is the planner fails to propagate trajectories through the voxel. Our solution is to treat high resolution subfaces as a single configuration point. For a given pair of high resolution subfaces, a trajectory is considered first between the centers of the two faces. If that test fails, off-center arcs of minimum turning radius are tested in an attempt to "connect" the two subfaces. This solution is less than perfect and still results in a loss of completeness. A better technique is needed.

The planning results for the real terrain meshes shown before are given in Figure 54. The first column lists the figure number of the terrain environment used. The second column indicates the position of the goal point. For the data in the first eight rows, the goal was positioned directly in front of the vehicle off the terrain. In the last eight rows the goal was positioned about six meters to the left of center off the terrain. Columns three through five list the number of voxels expanded, the CPU time in seconds required to compute the robot's kinematics (on a Sun 3/260), and the CPU time required to plan the trajectory (less the move time). The move time has been subtracted from the plan time since future systems will employ a lookup table reducing this time to near zero. The time required to construct the terrain pyramid from the polygonal mesh before planning is not shown since future systems will send

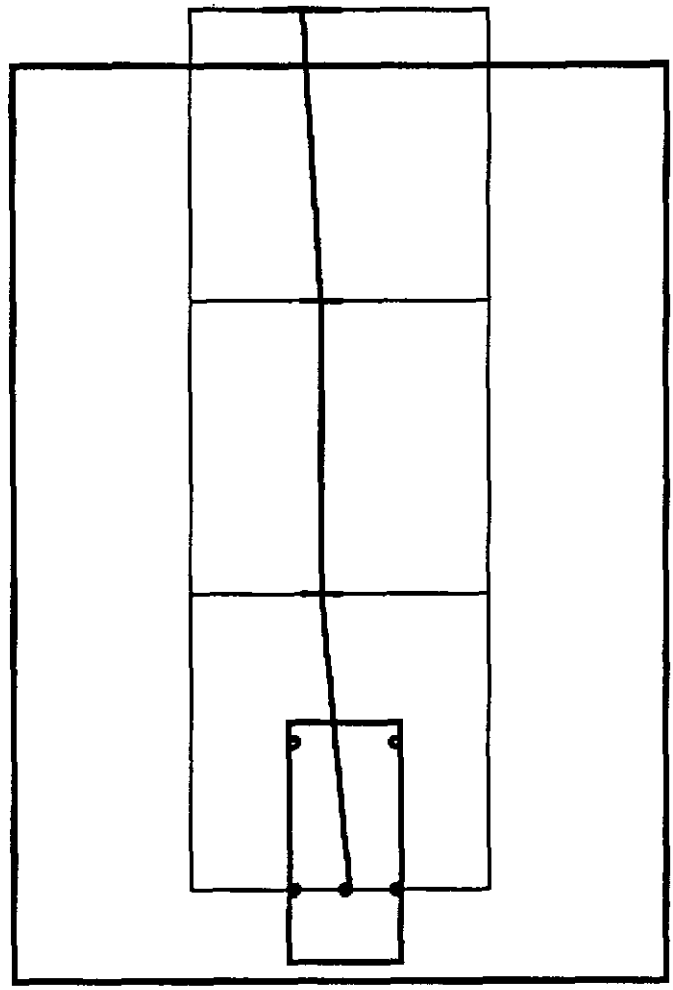


Figure 52: Planning Across Flat Terrain

images directly to the planner and the pyramid will be constructed in a constant three seconds. Due to the high cost of the polygonal intersection routine, the current time ranges from 5 to 200 seconds.

Note that the number of voxels expanded (and hence, planning time) varies as a function of the difficulty of the terrain. Planning time is increased both by environments that require high resolution voxels to resolve the constraints and by environments that require a convoluted solution trajectory. Rather than include figures for all sixteen planning problems, we've included a few that best illustrate the planning process. Figure 55 shows the environment in Figure 42 with the goal straight ahead. Note that this environment requires the expansion of only large voxels. Figure 56 shows the environment in Figure 46 with the goal left of center. In this case, the planner must find a path around the large inadmissible area on the left. Figure 57 shows the environment in Figure 48 with the goal straight ahead. In this case, the planner must find a path around the tree directly in front of the vehicle. Finally, Figure 58 shows the environment in Figure 49. In this case, even though the environment is quite complex, the planner needs only to test the largely admissible voxels in front of it to find a trajectory.

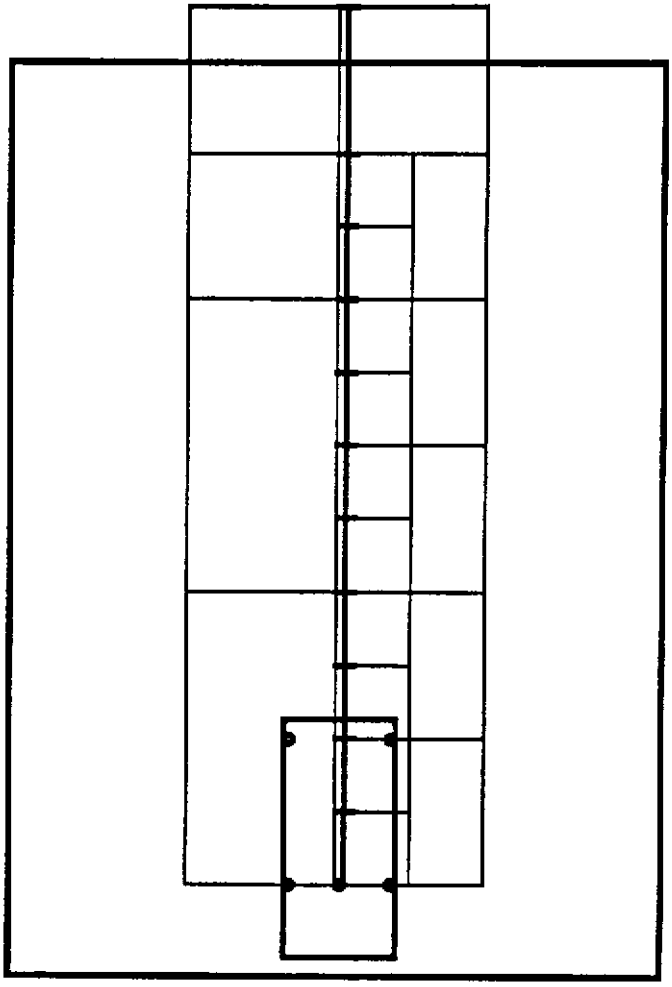


Figure 53: Planning Across Terrain Sloped at 5 Degrees

Figure	Goal	Voxels	Move Time	Plan Time
42	Straight	41	7	3.5
43	Straight	29	7	3
44	Straight	104	7	12
45	Straight	52	7	5.5
46	Straight	232	6	28
47	Straight	22	6	1.5
48	Straight	617	28	66
49	Straight	48	8	6
42	Turn	99	13	8.5
43	Turn	21	3	3
44	Turn	949	131	70
45	Turn	84	11	6.5
46	Turn	244	8	29
47	Turn	23	3	2
48	Turn	421	4	59
49	Turn	345	36	33

Figure 54: Planning Results for Real Terrain Data

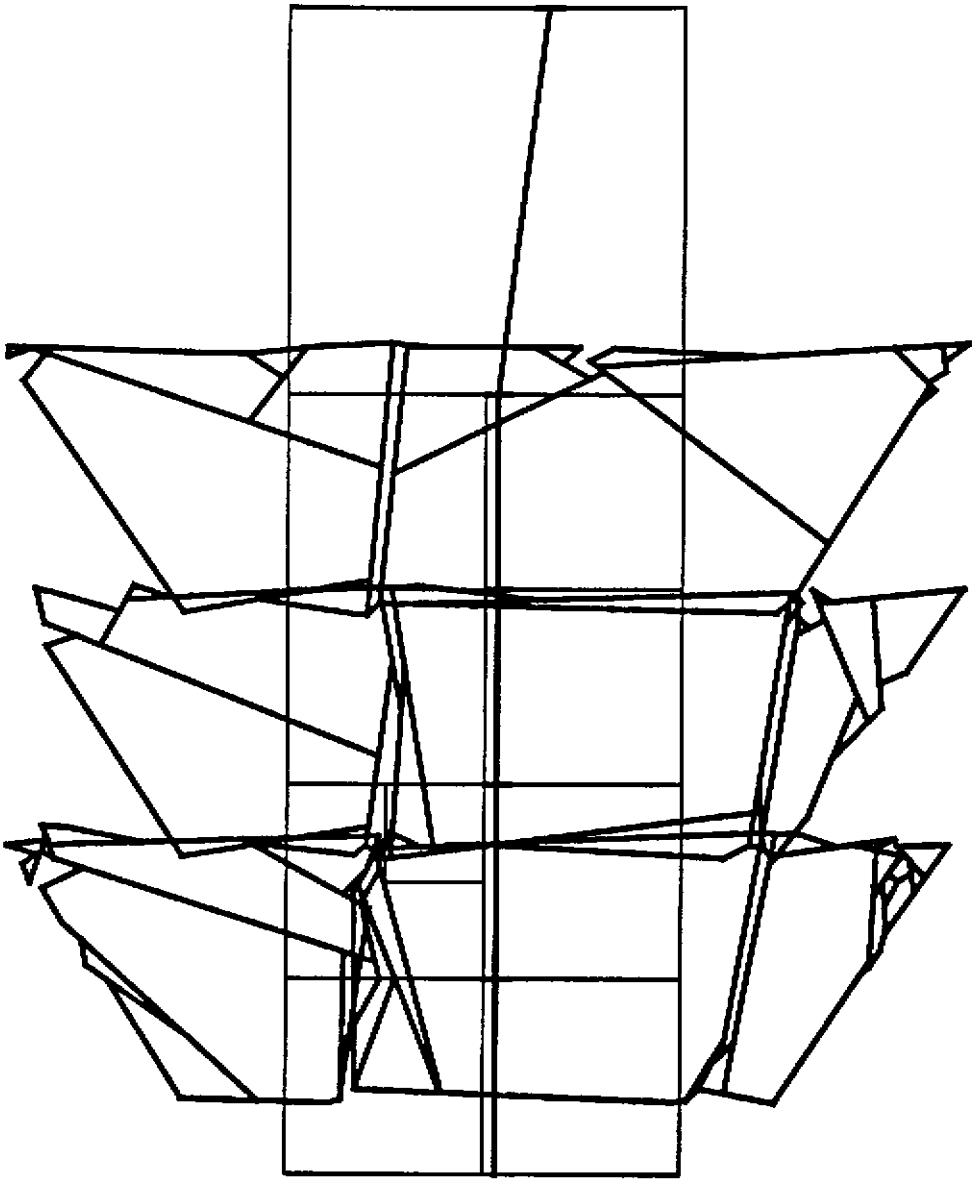


Figure 55: Planning through the Environment in Figure 42

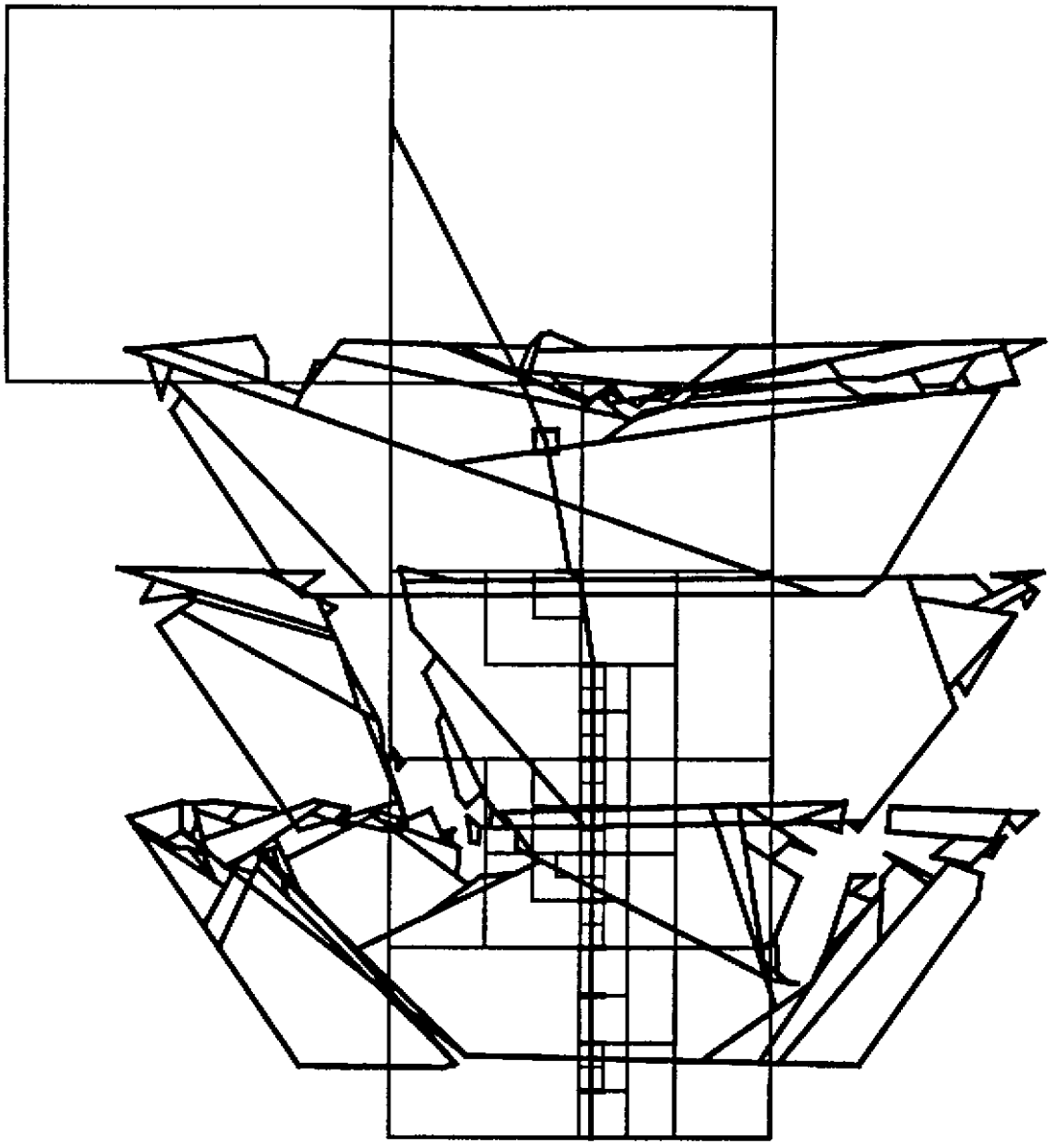


Figure 56: Planning through the Environment in Figure 46

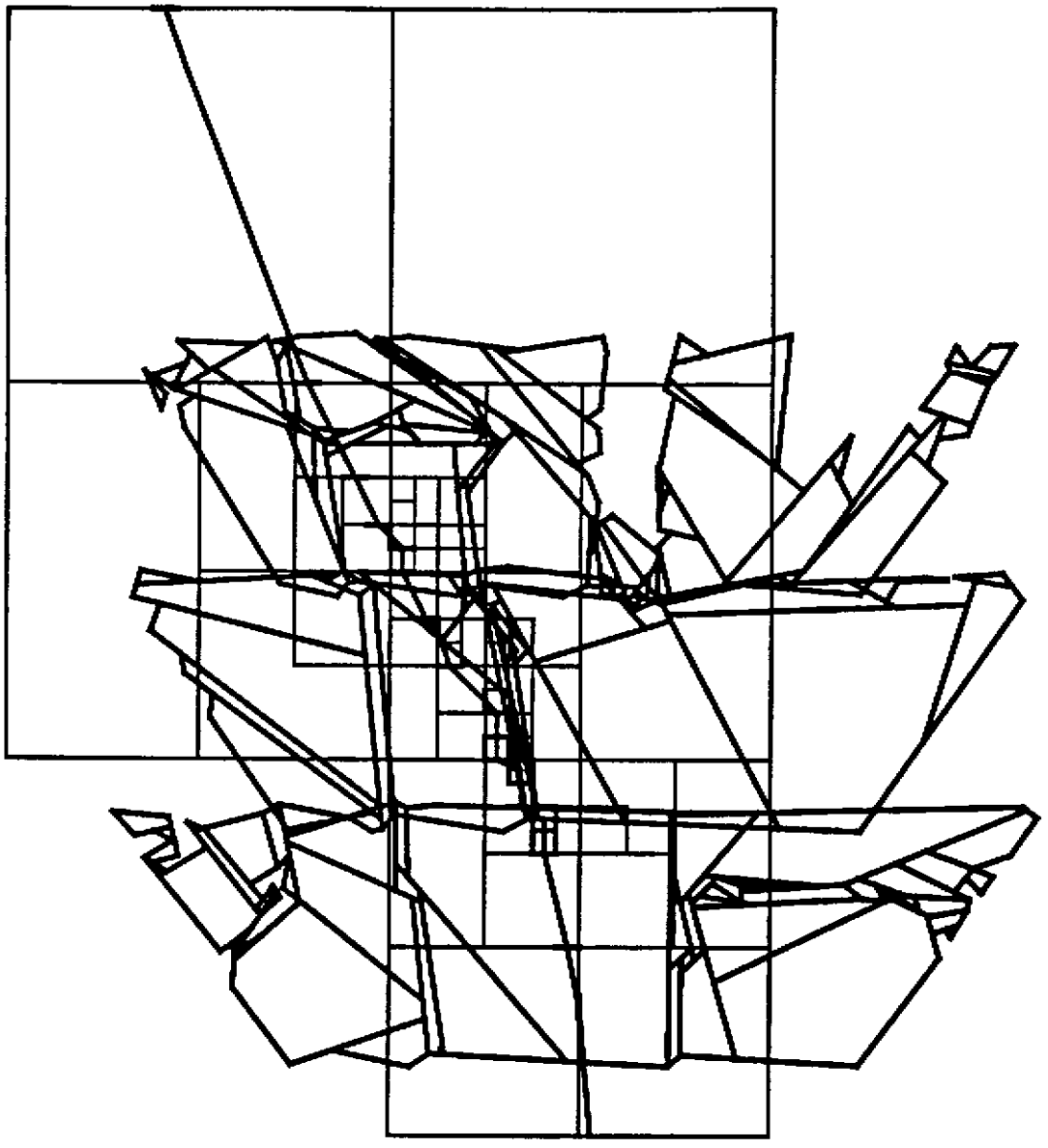


Figure 57: Planning through the Environment in Figure 48

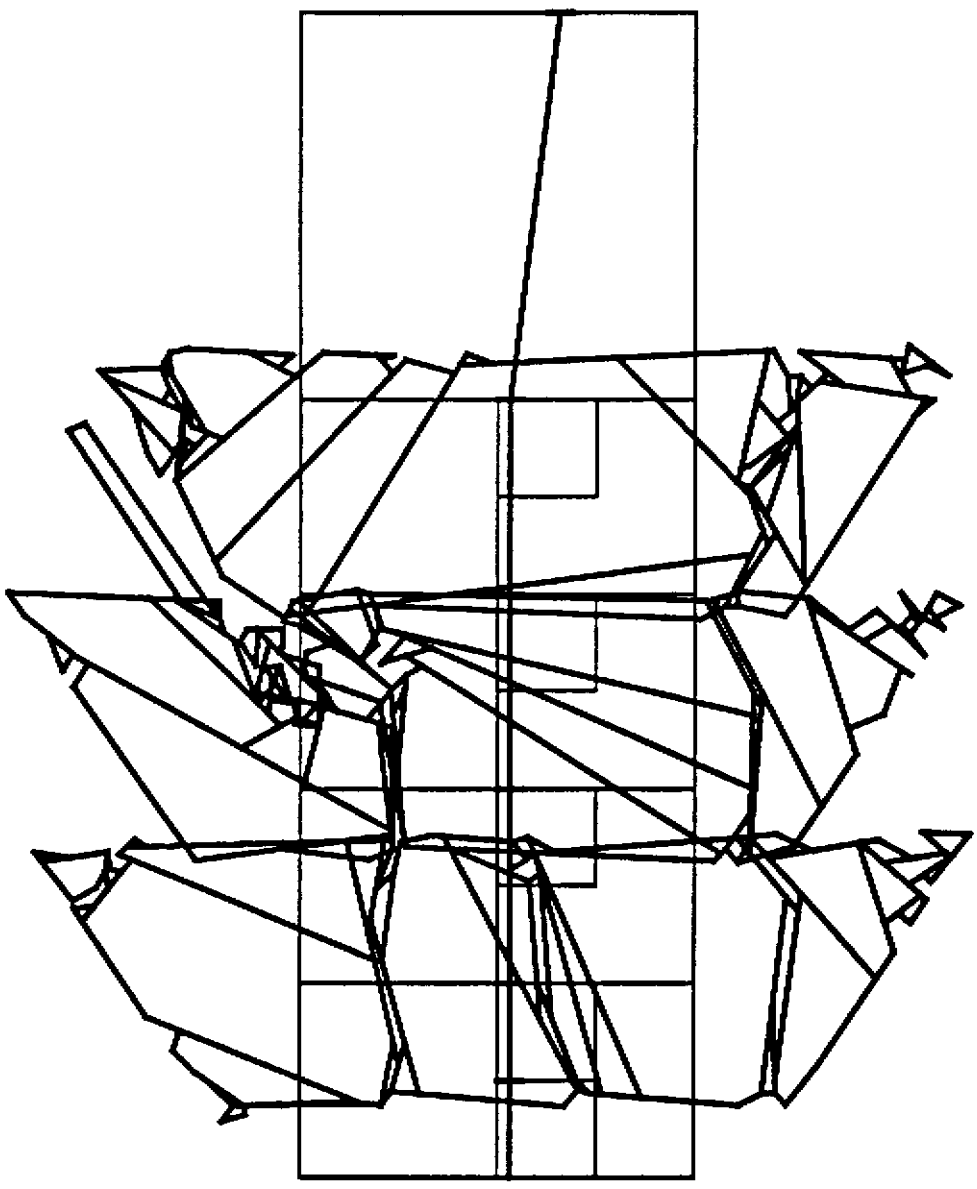


Figure 58: Planning through the Environment in Figure 49

Section IV
Conclusions

Evolution of the CODGER Blackboard and the Driving Pipeline Architecture

The CODGER system has been shown to be a good tool for developing a large robot navigation system. The "star-shaped" architecture allows any module to communicate with any other module. Although in the final system configuration the communication channels are typically less tightly coupled than *N* to *N*, CODGER was designed to be flexible by allowing the data-routing to change as the system evolves. The data format is flexible enough to represent a broad range of objects needed in a navigation system, including map information, physical objects, status reports, hypotheses, and commands. The system was designed so that changes in the data format affect only those modules which depend on the changes, thus avoiding recompilation of the entire system. This feature is especially important since large navigation systems typically include over one hundred thousand lines of code. Of special importance to navigation systems is geometric data. Sensing, planning, and control modules all need to use this type of data. CODGER provides mechanisms for representing and querying such data. It is able to automatically handle these operations, while accounting for the fact that such data is typically acquired at different times while the robot is moving.

CODGER is able to decrease the overall run-time for a navigation system by providing a means for distributing the computation across multiprocessors. Modules in the system are able to run in parallel, invoking a set of primitives in CODGER to synchronize the exchange of data between them. This mechanism is especially important for balancing the load between sensing and planning modules at the local navigation level. CODGER provides a locking mechanism to ensure data consistency between modules. A tree-based hierarchy of frames and frame generators is used to ensure consistency for geometric data. These mechanisms are needed to ensure that concurrently executing modules in the system operate with an updated and consistent model of the world at all times.

CODGER has a number of limitations. First, it is not a real-time system. Due to the relative long latencies in TCP/IP message passing and the varying execution patterns of processes in a UNIX time-sharing environment, data transfer cannot be guaranteed within given time bounds. This limitation is acceptable given the types of navigation scenarios we have addressed. For systems with real-time constraints, however, the lack of fast, direct communication channels makes CODGER inappropriate. Second, the low data transfer bandwidth of CODGER precludes the transfer of iconic or image-level data through the database. Again, the systems we have developed decompose the problem in such a way that high bandwidth is not required. Finally, the mapping of KS's or modules to processors in CODGER is static. CODGER is unable to redistribute modules on the processors to balance the load; this configuration must be set by the user a priori. We are able to compensate for this deficiency by adjusting the execution times of the various modules in the system to balance the load.

The NAVLAB architecture was demonstrated in several systems to be a suitable means for separating global, map-based navigation from the local sense-plan-drive cycle for moving the robot along a route. We developed a three-tiered map format that separated semantic, topological, and geometric data tagged accordingly with perceivable/navigable attributes. This format enabled components of the navigation system to digest only that portion of the map data necessary to perform its task. A system was demonstrated that started with a topological map of roads and intersections with sketchy metric data, and

filled in this data as it navigated.

The driving pipeline has been demonstrated as an effective way to increase the speed of the robot by taking advantage of a multi-processing environment. If parameters are adjusted correctly, the velocity of the robot can be increased by a factor equal to the ratio of the total processing time of all local operations divided by the stage time of the pipe. The linear sequencing of operations through the pipeline guarantees that the robot will not drive into an area that has not been cleared of all local operations. This feature enables the system to use general purpose computers (non-realtime systems) in a variety of navigation scenarios (where the time required by each stage can vary).

However, increased speed through pipelining comes at a cost. The Perception system must look out farther in front of the robot where data may be less reliable than in the stop-and-go case. The local planning space is reduced thus making the robot less responsive to its environment. The Driving Monitor lags behind Perception and the Environment Modeler, thus providing less accurate predictions to Perception.

Kinematic Path Planning for Wheeled Vehicles

A mobile robot such as the NAVLAB in rough terrain can be adequately represented by the composition of the elevation values of the terrain with three configuration space parameters, (x, y, θ). Constraint inequalities defined over the terrain function can be constructed to determine admissibility for a mobile robot. Examining the upper and lower bounds of each constraint is an efficient way of determining the admissibility of a subspace of configurations. The octree is an efficient way to construct the subspace of configurations representing admissibility over the entire configuration space. Finally, a pyramid of elevations at multiple resolutions is an efficient representation for the terrain function.

We have identified five criteria for evaluating a path planner: admissibility model, soundness, completeness, optimality, and complexity. Since the definition of admissibility in unstructured terrain can be complex, it is imperative that a planning paradigm, such as the tesselation-based approach, that supports approximations to the admissible space be employed. Existing tesselation approaches do not adequately guarantee trajectory soundness. We have extended this approach to take kinematic constraints into consideration. In the process we sacrifice optimality, which for outdoor mobile robots is the least important of the five criteria.

In order to integrate a local path planner into a larger system, the planner must be able to handle goal specifications more general than a single configuration point. We have developed techniques for planning within local boundaries, for targeting goals not within the search space, and for planning with goal subspaces corresponding to the set of possible vantage points for a landmark, or coverage points for off-road navigation.

Once a coarse trajectory has been planned from start to goal, we have shown how to smooth the trajectory without violating the path soundness. Gradient-descent techniques such as path relaxation can be used to smooth the trajectory, provided that local minima in the smoothing function can be avoided. If not, a global technique such as dynamic programming must be employed.

Future Directions

This is the final report for the current contract, so it marks the termination of the current research program. However, we have identified several topics that we believe are important areas for further research in the general area of mobile robot vehicles.

As we have gained more experience with the CODGER blackboard and the tightly synchronized Driving Pipeline, we have seen the need to look beyond their horizons to the next evolutionary steps in mobile robot architectures. For roadway driving, the Driving Pipeline has been very useful in implementing continuous motion. However, it requires that the color vision for road-following and range data interpretation for obstacle detection must work in complete synchronization. This is inefficient, since the two different sensors have different fields of view and different perceptual complexities. Therefore, we hope to continue research with the NAVLAB that moves away from this tightly sychronized architecture towards one that allows each sensor system to work at its own rate.

At the same time, we believe we have identified a substantial subsystem of the NAVLAB that is currently implemented with the blackboard but ought to be implemented differently. This is the lower-level set of modules that performs obstacle detection by range data analysis, path planning, and monitoring of the vehicle's actual motion. We are now planning a new system we call EDDIE, in which these low-level modules would form a subsystem with highly optimized communication channels, while the CODGER blackboard would be used by the higher-level planning modules to communicate with each other and with this low-level subsystem.

In EDDIE, we would like to incorporate our new kinematic path planner, which is presented in this report. It has so far been implemented and tested on real terrain data from the ERIM laser scanner, but it has not yet been run on the vehicle itself. We would like to test it on the vehicle first as a replacement for our current path planning module, then use it as one of the key elements of the EDDIE system.

References

[1] Aho, A. V., Hopcroft, J. E., Ullman, J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley Publishing Company, 1974.

[2] Alleva, F., Bisiani, R., Forin, A., Lerner, R.
AGORA Distributed, Interactive Speech Kernel.
1985.

[3] Bekker, M. G.
The Theory of Land Locomotion.
The University of Michigan Press, 1956.

[4] Bekker, M. G.
Off-the-Road Locomotion.
The University of Michigan Press, 1960.

[5] Bledsoe, W. W.
The Sup-Inf Method in Presburger Arithmetic.
Technical Report, University of Texas at Austin, 1974.
Memo ATP 18.

[6] Brooks, R. A.
Symbolic Reasoning Among 3-D Models and 2-D Images.
PhD thesis, Stanford University, June, 1981.

[7] Brooks, R. A.
Solving the Find-Path Problem by Representing Free Space as Generalized Cones.
Technical Report, Massachusetts Institute of Technology, May, 1982.
A. I. Memo No. 674.

[8] Canny, J.
Finding Edges and Lines in Images.
Technical Report, Massachusetts Institute of Technology, June, 1983.
Master's thesis.

[9] Canny, J.
The Complexity of Robot Motion Planning.
PhD thesis, Massachusetts Institute of Technology, May, 1987.

[10] Chatila, R., Laumond, J-P.
Referencing and Consistent World Modelling for Mobile Robots.
In *Proc. of the IEEE on Robotics and Automation.* March, 1985.

[11] Crowley, J.
Navigation for an Intelligent Mobile Robot.
Technical Report, CMU Robotics Institute, 1984.
CMU-RI-TR-84-18.

[12] Daily, M., Harris, J., Keirsey, D., Olin, K., Payton, D., Reiser K., Rosenblatt, J., Tseng, D., Wong, V.
Autonomous Cross-Country Navigation with the ALV.
In *Proc. IEEE International Conference on Robotics and Automation.* 1988.

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

[14] Elfes, A. E.
Sonar Navigation.
In *Workshop on Robotics.* Oak Ridge National Lab, Oak Ridge TN, August, 1985.

[15] Elfes, A.
Sonar-Based Real-World Mapping and Navigation.
IEEE Journal of Robotics and Automation 3(3), June, 1987.

[16] Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R.
The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty.
ACM Computing Surveys 12(2), June, 1980.

[17] Faverjon, B.
Obstacle Avoidance Using an Octree in the Configuration Space of a Manipulator.
In *Proc. IEEE International Conference on Robotics.* March, 1984.

[18] Fujimura, K., Samet, H.
A Hierarchical Strategy for Path Planning Among Moving Obstacles.
IEEE Transactions on Robotics and Automation 5(1), February, 1989.

[19] Gelernter, D., Carriero, N., Chandran, S., Chang, S.
Parallel Programming in Linda.
Technical Report, Yale University, January, 1985.
YALEU/DCS/RR-359.

[20] Giralt, G., Sobek, R., Chatila, R.
A Multi-Level Planning and Navigation System for a Mobile Robot; A First Approach to Hilare.
In *Proc. IJCAI-79.* August, 1979.

[21] Giralt, G., Chatila, R., Vaisset, M.
An Integrated Navigation and Motion Control System for Autonomous Multisensory Mobile Robots.
In *First International Symposium on Robotics Research.* 1983.

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

[23] Goto, Y., Stentz, A.
Mobile Robot Navigation: The CMU System.
IEEE Expert 2(4), Winter, 1987.

[24] Goto, Y., Shafer, S., Stentz, A.
The Driving Pipeline: A Driving Control Scheme for Mobile Robots.
International Journal of Robotics and Automation 4(1), 1989.

[25] Harmon, S.
USMC Ground Surveillance Robot: A Testbed for Autonomous Vehicle Research.
In *Proc. of the Fourth UAH/UAB Robotics Conference.* April, 1984.

[26] Harmon, S. Y.
Implementation of Complex Robot Subsystems on Distributed Computing Resources.
Technical Report, Naval Ocean Systems Center, 1986.

[27] Hebert, M., Kanade, T.
First Results on Outdoor Scene Analysis.
In *Proc. IEEE International Conference on Robotics and Automation.* 1985.

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

[29] Hebert, M., Kanade, T.
3-D Vision for Outdoor Navigation by an Autonomous Vehicle.
In *Proc. Image Understanding Workshop.* 1988.

[30] Herman, M.
Fast, Three-Dimensional, Collision-Free Motion Planning.
In *Proc. IEEE International Conference on Robotics and Automation.* April, 1986.

[31] Jackins, C. L., Tanimoto, S. L.
Oct-Trees and Their Use in Representing Three-Dimensional Objects.
Computer Vision, Graphics, and Image Processing 14(3), 1980.

[32] Khatib, O.
Real-Time Obstacle Avoidance for Manipulators and Mobile Robots.
The International Journal of Robotics Research 5(1), Spring, 1986.

[33] Korf, R. E.
Real-Time Heuristic Search.
1987.
Submitted to Artificial Intelligence.

[34] Krogh, B.
A Generalized Potential Approach to Obstacle Avoidance Control.
In *Proc. Robotics Research Conference.* August, 1984.

[35] Laumond, J-P.
Finding Collision-Free Smooth Trajectories for a Non-Holonomic Mobile Robot.
In *Proc. IJCAI-87.* August, 1987.

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

[37] Lozano-Perez, T.
Automatic Planning of Manipulator Transfer Movements.
IEEE Transactions on Systems, Man, and Cybernetics SMC-11(10), October, 1981.

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

[39] McTamaney, L.
Real-Time Intelligent Control.
IEEE Expert 2(4), Winter, 1987.

[40] Moravec, H. P.
The Stanford Cart and the CMU Rover.
Proc. IEEE 71(7), July, 1983.

[41] Nii, H. Penny.
Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, Part One.
AI Magazine 7(2):38-53, Summer, 1986.

[42] Nii, H. Penny.
Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, Part Two.
AI Magazine 7(3), Fall, 1986.
Forthcoming.

[43] Nilsson, N. J.
A Mobile Automaton: an Application of Artificial Intelligence Techniques.
In *Proc. IJCAI-69.* May, 1969.

[44] Nilsson, N. J.
Principles of Artificial Intelligence.
Tioga Publishing Company, 1980.

[45] Preparata, F. P., Shamos, M. I.
Computational Geometry: An Introduction.
Springer-Verlag, 1985.

[46] 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.

[47] Sharir, M., Schorr, A.
On Shortest Paths in Polyhedral Spaces.
In *Proc. 16th ACM STOC.* 1984.

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

[49] Thompson, A. M.
The Navigation System of the JPL Robot.
In *Proc. IJCAI-77.* 1977.

[50] Thorpe, C. E.
FIDO: Vision and Navigation for a Mobile Robot.
PhD thesis, CMU Computer Science Dept., December, 1984.

[51] Turk, M. A., Morgenthaler, D. G., Gremban, K. D., Marra, M.
VITS - A Vision System for Autonomous Land Vehicle Navigation.
IEEE Transactions on Pattern Analysis and Machine Intelligence 10(3), May, 1988.

Publications

Stentz, A., "The NAVLAB System for Mobile Robot Navigation", Ph.D. Thesis, Carnegie Mellon University Computer Science Dept., November 1989.

Stentz, A., "Multi-Resolution Constraint Modeling for Mobile Robot Planning", Proceedings of SPIE Symposium on Advances in Intelligent Robotics Systems, November 1989.