DTIC FILE COPY

AD-A214 181

NPS52-89-040

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

DTIC
ELECTE
NOV 08 1989
S D
E

PLANNING MINIMUM-ENERGY PATHS IN AN OFF-ROAD ENVIRONMENT WITH ANISOTROPIC TRAVERSAL COSTS AND MOTION CONSTRAINTS

by

Ron S. Ross

June 1989

Dissertation Supervisor: Robert B. McGhee

Approved for public release; distribution is unlimited.

Prepared for:
Naval Postgraduate School
Monterey, CA. 93943-5000

89 11 06 160

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Harrison Schull
Provost

This thesis is prepared in conjunction with research sponsored in part by contract from the United States Army TEXCOM Experimentation Center (USATEC) under MIPR ATEC 88-86.

Reproduction of all or part of this report is authorized.

This thesis is issued as a technical report with the concurrence of:

ROBERT B. MCGHEE
Professor and Chairman
of Computer Science

Reviewed by:

ROBERT B. MCGHEE
Professor and Chairman
Department of Computer Science

Released by:

KNEALE T. MARSHALL
Dean of Information and
Policy Sciences

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-89-040	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*) Code 52	7a. NAME OF MONITORING ORGANIZATION U.S. Army TEXCOM Experimentation Center
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Fort Ord, CA 93941
8a. NAME OF FUNDING / SPONSORING ORGANIZATION U. S. Army TEXCOM Experimentation Center 8b OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MIPR ATEC 88-86
8c. ADDRESS (*City, State, and ZIP Code*) Fort Ord, CA 93941	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (*Include Security Classification*)
PLANNING MINIMUM-ENERGY PATHS IN AN OFF-ROAD ENVIRONMENT WITH ANISOTROPIC TRAVERSAL COSTS AND MOTION CONSTRAINTS

12. PERSONAL AUTHOR(S)
Ross, Ron S.

13a TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15 PAGE COUNT
Ph.D. Dissertation	FROM _____ TO _____	1989 June	278

16 SUPPLEMENTARY NOTATION
The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Path planning, Obstacle avoidance, Search, Mobile robots, Knowledge representation, Spatial reasoning, Route planning

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

For a vehicle operating across arbitrarily-contoured terrain, finding the most fuel-efficient route between two points can be viewed as a high-level global path-planning problem with traversal costs and stability dependent on the direction of travel (anisotropic). The problem assumes a two-dimensional polygonal map of homogeneous cost regions for terrain representation constructed from elevation information. The anisotropic energy cost of vehicle motion has a non-braking component dependent on horizontal distance, a braking component dependent on vertical distance, and a constant path-independent component. The behavior of minimum-energy paths is then proved to be restricted to a small, but optimal set of traversal types. An optimal-path-planning algorithm, using a heuristic search technique, reduces the infinite number of paths between the start and goal points to a finite number by generating sequences of "goal-feasible" window lists from analyzing the polygonal map and applying pruning criteria. The pruning criteria consist of visibility analysis, heading analysis, and region-boundary constraints. Each goal-feasible window list specifies an associated convex optimization problem, and the best of all locally-optimal paths through the goal-feasible window lists is the globally-optimal path. These ideas have been implemented in a computer program, with results showing considerably better performance than the exponential average-case behavior predicted.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Robert B. McGhee	22b TELEPHONE (*Include Area Code*) (408) 646-2449
	22c OFFICE SYMBOL Code 52Mz

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office 1986—606-243

Approved for public release, distribution unlimited

PLANNING MINIMUM-ENERGY PATHS IN AN OFF-ROAD ENVIRONMENT WITH ANISOTROPIC TRAVERSAL COSTS AND MOTION CONSTRAINTS

by

Ron S. Ross
Major, United States Army
B.S., United States Military Academy, 1973
M.S., Naval Postgraduate School, 1982

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

Author: Ron S. Ross
Ron S. Ross

Approved by:

C. Thomas Wu
Associate Professor of Computer Science

Harold M. Fredricksen
Professor of Mathematics

Edward B. Rockower
Associate Professor of Operations Research

Neil C. Rowe
Associate Professor of Computer Science

Michael J. Zyda
Associate Professor of Computer Science

Robert B. McGhee
Professor of Computer Science
Dissertation Supervisor

Approved by:
Robert B. McGhee, Chairman, Department of Computer Science

Approved by: K. T. Marshall
Kneale T. Marshall, Dean, Information and Policy Sciences

ABSTRACT

For a vehicle operating across arbitrarily-contoured terrain, finding the most fuel-efficient route between two points can be viewed as a high-level global path-planning problem with traversal costs and stability dependent on the direction of travel (anisotropic). The problem assumes a two-dimensional polygonal map of homogeneous cost regions for terrain representation constructed from elevation information. The anisotropic energy cost of vehicle motion has a non-braking component dependent on horizontal distance, a braking component dependent on vertical distance, and a constant path-independent component. The behavior of minimum-energy paths is then proved to be restricted to a small, but optimal set of traversal types. An optimal-path-planning algorithm, using a heuristic search technique, reduces the infinite number of paths between the start and goal points to a finite number by generating sequences of "goal-feasible" window lists from analyzing the polygonal map and applying pruning criteria. The pruning criteria consist of visibility analysis, heading analysis, and region-boundary constraints. Each goal-feasible window list specifies an associated convex optimization problem, and the best of all locally-optimal paths through the goal-feasible window lists is the globally-optimal path. These ideas have been implemented in a computer program, with results showing considerably better performance than the exponential average-case behavior predicted.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and thanks to the many persons who helped make this dissertation possible. I am especially grateful to my advisor, Professor Robert B. McGhee for his constant motivation, support, and encouragement throughout the work and to Professor Neil C. Rowe for his valuable insights on path-planning issues. I would also like to thank all of the members of my PhD. committee for their careful review of the manuscript and helpful comments.

And finally, to my wife Conni and our two children Kip and Katie, a special note of thanks for all of your love, understanding and support. You made it all worthwhile.

I. INTRODUCTION

A. GENERAL BACKGROUND

Recent technological advances in robotics and robotic vehicles have generated renewed interest within the artificial intelligence and operations research communities in developing new solutions to traditional path-planning and obstacle-avoidance problems. Operating a *mobile robot*, in an off-road environment across arbitrarily-contoured terrain presents a path-planning problem that is inherently difficult to solve. The degree of difficulty is dependent on the quantity and quality of *knowledge* available to the problem solver. In the context of the terrain navigation problem, it is assumed here that a mobile robot has access to a database of *a priori* knowledge about the environment in the form of a terrain map. The path-planning problem considered in this dissertation only considers high-level, global planning tasks.

The navigational tasks that must be accomplished by a mobile robot are clarified by examining the human analog. Cartographic maps allow human beings to (1) establish a current position on the map and (2) plan and execute routes between two locations. The specific skills required to accomplish these tasks derive from the complex process of *spatial reasoning*, that is, reasoning about the physical properties of objects on the map, including shape, position, and motion [Ref. 1]. Spatial reasoning for paths means not violating any physical constraints, avoiding obstacles, and bypassing areas that present a clear stability danger to the vehicle

Finding the "best" path implies an optimization of the *cost* of movement along the path according to a specified criterion. The cost can include distance, time, or any other relevant factor. In this dissertation, the traversal cost for a mobile robot is expressed in terms of *energy*. Energy expenditure is directly related to fuel consumption, and minimum-energy paths are the most fuel-efficient, not necessarily the shortest.

Optimal-path-planning techniques employ the problem-solving paradigm of *search* using a set of abstract descriptions of possible actions [Ref. 2]. There are three subproblems. First, there must be an appropriate mathematical model to describe the physical relationship between the mobile robot and the natural terrain. Second, there must be appropriate techniques to extract and represent important terrain properties such as geometric configuration and surface composition. Finally, there must be an efficient path-planning algorithm or search strategy to generate the optimal path.

As for the robot-to-terrain mathematical model, most research assumes *isotropic* terrain and motion costs independent of the direction of travel. This is not adequate for energy-based path-planning problems since the motion costs for a vehicle on sloped terrain are related to its heading (azimuth) or are *anisotropic*. This can be observed by noting the change in the inclination angle of the vehicle as it assumes a range of possible headings from the steepest (gradient) to a level curve (contour). The differences in the inclination angle directly affect energy costs. Furthermore, certain headings can be inherently unsafe from the standpoint of vehicle *stability*, something that must be considered in finding a path. Although terrain cost data is approximate, the averaging of anisotropic effects into an isotropic model is undesirable since it cannot account for the "heading-specific" differences in energy costs as well as the ranges of headings that are impermissible for reasons of stability.

This dissertation does not contribute to the problem of terrain representation, but it is assumed the terrain surface can be modeled as an irregular *polyhedron*. In general, polyhedral models are more difficult to obtain than grid-based models. A polyhedral model is a collection of interconnected convex planar faces representing the aggregation of groups of contiguous, gridded data points that have the property of constant gradient within some designated threshold. It can be advantageous to use a polyhedral representation since it facilitates reasoning about "terrain regions" instead of individual "points" on the map. This type of representation is more efficient with respect to storage space and can be a more effective structure to search if the number of regions does not become large. Path traversals are not restricted to the nearest neighboring grid points and therefore, the error introduced through digitization bias is eliminated.

A two-dimensional representation can be derived from the polyhedral model by projecting each polygonal face onto the cartographic map plane giving a *polygonal mesh* or irregular tessellation of the two-dimensional map plane [Ref. 3]. Each polygon within the mesh can be further subdivided into regions of homogeneous characteristics, such as uniform soil type and vegetation. Such reduction of three-dimensional terrain information to a two-dimensional map plane is described by Gaw and Meystel [Ref. 4] as a *two-and-one-half-dimensional* terrain representation. Although polygonal models seem to have unnatural discontinuities at boundaries and hence, can be accused of being a poor representation of the real world, this concern is primarily an artifact of the process that constructs the polygonal regions. Higher-resolution gridded data sets to produce polyhedrons more closely resembling the natural terrain can always be produced.

The path-planning model described in this dissertation exploits the fact that there are only certain ways an optimal path can cross the polygonal mesh boundaries. Thus, an infinity of possible paths between the start and goal points can be reduced to a finite, but provably optimal, set of path possibilities. This is accomplished by defining a set of "equivalence classes" or sequences of vertices and edges through which an optimal path must pass enroute from the start to the goal. Path-possibility "pruning" can occur as a result of simple geometric visibility analysis as illustrated in Figure 1.1 or as a result of certain stability constraints that, if violated, would cause the vehicle to overturn. Having reduced the problem to a finite set of possibilities, a *heuristic* search algorithm (modified A^*) can efficiently evaluate the alternatives to find the minimum-energy path. The search is heuristic in that information "prunes" or eliminates unproductive paths as early as possible in the planning process [Ref. 5]. The optimal path consists of a set of piecewise-linear segments across the two-dimensional polygonal mesh.

An alternative search technique known as wavefront propagation, applies omnidirectional, uniform-cost search (the dynamic programming paradigm) to a uniform grid of data points to find optimal paths. This approach is described in detail by Richbourg [Ref. 6]. In certain situations the wavefront-propagation method is clearly inferior to a heuristic method of search using symbolic terrain. The wavefront-

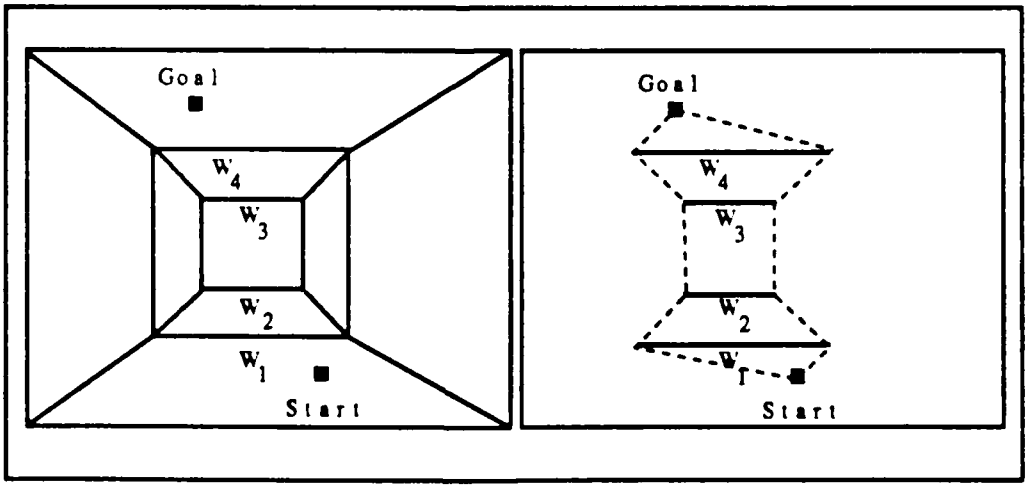

Figure 1.1 Path-possibility Pruning

propagation technique is less desirable when accuracy of the final solution is of prime importance; the information content of the terrain is compromised by the imposition of a uniform grid at an arbitrary resolution, resulting in an inherent error due to digitization bias [Ref. 7]. If the number of regions (and therefore potential boundary crossings) does not become large, the heuristic search across symbolic terrain can offer a more time-efficient path-planning alternative [Ref. 6]. This assumes that the exponential worst-case behavior can be improved significantly by the use of various "pruning criteria" that effectively reduce the search space. The symbolic representation can also prove to be the most space-efficient, especially for large map areas with gradual changes in terrain.‡

In addition to the wavefront propagation method, the calculus of variations offers a fully general approach to path-planning problems [Ref. 9, 10]. However, this technique is not appropriate for the minimum-energy path-planning problem because of the discontinuities in the path space caused by the polyhedral boundaries. Also, to avoid convergence to a local minimum, the calculus-of-variations approach requires a reasonable approximation to the actual optimal path as an input variable. This is impractical inasmuch as the computational cost of a "good" initial approximation to the true optimal path may approach the cost of solving the path-planning problem.

A final aspect of the path-planning problem that must be considered is error. There are two potential sources. The original gridded data set can contain a significant error factor depending on the procedures used for collecting, interpreting, and assembling the terrain information. This source of error is not considered in this dissertation; it is assumed that the polyhedral model is constructed from "perfect data". Another source of error results from the actual construction of the irregular polyhedron from the uniform gridded data. The error in each planar face can occur in both slope and orientation and is considered a second order effect on the final optimal path solution.

B. ORGANIZATION

Chapter II presents a comprehensive survey of the previous research conducted in the three principal areas relevant to this dissertation: vehicle mobility models, terrain representation, and path-planning techniques. Chapter III introduces the mathematical model of the interaction between vehicular systems and natural terrain surfaces. A symbolic terrain representation is also proposed in Chapter III. Chapter IV

---

‡ Wavefront methods can be extended to problems with anisotropic cost functions as demonstrated by Parodi [Ref. 8].

describes the optimal-path-planning model. The ideas of Chapters III and IV form the basis of the computer program described and evaluated in Chapter V. Chapter VI summarizes the dissertation.

II. SURVEY OF LITERATURE

This literature survey includes a review of previous research conducted in the areas of terrain representation, vehicle mobility, and path planning. Particular emphasis is placed on vehicle mobility and path planning as the two principal areas of contribution for this dissertation. Terrain representations are examined only to gain a frame of reference for the types of symbolic objects used in mobility analysis and automatic path-planning problems.

A. TERRAIN REPRESENTATION

1. Low-Level Terrain Representations

Arbitrarily-contoured terrain, which is generally continuous in nature, must be discretized in order to be used by mobility models and path-planning algorithms that are operating on digital computers. Thus, it is necessary to decide what type of terrain information should be made explicit. There are three fundamental classes of terrain information important to vehicle mobility and path-planning problems: (1) surface configuration, (2) surface composition, and (3) surface covering. Surface configuration refers to the geometric structure of the terrain. Surface composition relates to the type of ground materials that comprise the terrain area. The kinds of objects that are present on top of the terrain such as obstacles and vegetation are part of the surface covering class. Associated with each of these classes is a set of individual *terrain factors*, or attributes, that describes a particular aspect of the terrain [Ref. 11]. Although the complete set of terrain factors is extensive, only a select few are relevant to the minimum-energy path-planning problem. For example, the terrain factors of interest for the configuration class are *slope* and *orientation*. For surface composition, the key terrain factors are *soil strength* and *soil type*. The process of discretization requires sampling the terrain at various points and quantifying the attributes at those points. The resulting terrain representation is defined as a *low-level representation* because it describes the terrain surface at the individual "data point" level and does not recognize any relationships or connectivity between points.

The Defense Mapping Agency (DMA) is the primary source of digital cartographic information. The Digital Landmass System (DLMS) is the standard, multi-use terrain database for digital mapping, charting and geodetic products [Ref. 12]. There are two standard approaches to representing terrain data in digital form. The first approach represents the particular terrain attributes as discrete data

points organized in a *uniform grid*. The Defense Mapping Agency (DMA) maintains a database of elevations in digital form according to the gridded data format. The Digital Terrain Elevation Database (DTED) provides elevation data at varying degrees of *resolution* depending on the requirements of the particular application. Typically, the elevation data is produced at a low resolution of three arc second intervals, or approximately 100 meters. For selected areas, higher resolution data are also available using one arc second intervals, or approximately 25 meters [Ref. 13].

The U. S. Army Engineer Waterways Experiment Station (WES) has developed a uniform gridded data representation focusing on a descriptive entity called the *terrain unit*. Six terrain factors are selected as the attributes of interest: (1) slope, (2) vegetation type, (3) stem spacing, (4) stem diameter, (5) surface material type, and (6) depth of surface. The grid is scanned and a separate terrain unit number is assigned to each unique combination of factor values. The terrain unit number and the corresponding elevation value for each grid point is recorded for subsequent analysis [Ref. 11].

An alternative to the grid approach attempts to represent aggregations of data points with similar attribute values as homogeneous *regions*. Each region is described geometrically by a convex polygon consisting of a finite set of vertices and edges and functionally by "attaching" descriptive attributes to the region. DMA produces a second digital database of terrain feature information using the polygonal representation. The Digital Feature Analysis Database (DFAD) contains cultural feature data for a variety of terrain factors to include forest regions, lakes, rivers, road networks, and obstacles. The linear features, such as roads and rivers, are represented in the database by a set of connected line segments.

The process of combining similar attribute values can be relatively straightforward or extremely complex, depending on the type of attributes involved. For attributes such as vegetation or soil type, a simple region-growing or edge-finding technique as described in [Ref. 14] can be employed to create groups of contiguous data points. Polygonal boundaries can be fitted to the regions, and, if the region is concave, a splitting algorithm can be applied to generate a set of unique convex regions [Ref. 15]. For surface configuration attributes, the process is more difficult. Creating a three-dimensional polyhedron from basic elevation data points requires aggregating data points with similar gradient values, that is, equal slopes and orientations within some designated *threshold*. The object is to fit a set of data points to a particular plane with the constraint that the intersecting planes form a polygonal mesh.

The problem of producing the individual planar faces of a polyhedron, or set of convex polygons arranged in a geometric mesh has been explored by Rowe and Yee [Ref. 16]. Several approaches

are used to generate the planar patches. The first algorithm employs a top-down, quadtree-subdivision method followed by a bottom-up merging of similar subregions. Alternatively, a strict bottom-up approach with and without data smoothing, relies on conventional region growing techniques to develop the planar patches.

The process of generating *polygonal* terrain can be viewed in a hierarchic manner. At the lowest level, every pair of data points can be connected which, essentially, triangulates the terrain surface. The high-resolution terrain model proposed by Zyda [Ref. 17] employs this representational method. Although conceptually simple, the triangulation method produces the maximum number of convex polygons tiling the polyhedral structure. The next level of the hierarchy proposes combining the triangles with equal gradients within a designated threshold or tolerance. In essence, the triangle *primitives* combine to form larger convex polygonal regions with similar characteristics. The object is to generate the minimum number of convex polygonal regions covering the polyhedron so each region maintains the *constant gradient* property within the designated threshold, and the boundaries form a geometrically consistent polygonal mesh.

2. Alternative Terrain Representations

Low-level terrain representations are limited in descriptive power due to the fact that there is little, if any, information on *functional* or *spatial* relationships between terrain features. Functional relationships are defined by Kwan [Ref. 18] as a hierarchy of terrain objects and classes of objects. With this representation, a terrain area can be described by a tree structure with each successive level of the tree defining a terrain object in greater detail. For example, the first level of the tree may contain the functional terrain class of vegetation. The next level of the tree partitions the vegetation entry into its valid subclasses, e.g., a forest, scrub, or swamp. Subsequent levels in the hierarchy expand the description of the subclass entries. The forest entry may contain information on the type of forest, stem spacing, stem diameter, and any other information relevant to the application. The hierarchy is analogous to the representations found in many artificial intelligence systems where information can be *inherited* through links in the tree using the *is-a* and *a-kind-of* relationships [Ref. 1].

Spatial relationships describe the connectivity properties of terrain features. A mixed representation of free space, defined by Kwan, Zamiska, and Brooks [Ref. 19] divides the terrain into two basic shape primitives: (1) convex polygons and (2) generalized cones. A connectivity graph is then developed to describe the topological relationships between obstacles. The connectivity graph facilitates

the development of corridors between obstacle regions. These corridors or "channels" are constructed using the generalized cone shape primitive. All remaining free space is described by convex polygons designated as "passage regions". The objective of the mixed free space approach is to provide a symbolic, higher-level map representation for spatial reasoning tasks such as path planning.

A spatial database management system proposed by Antony and Emmerman [Ref. 20] builds a terrain representation based on the *region-quadtree* approach of Samet [Ref. 21] and the *frame* structure of Minsky [Ref. 22]. The region quadtree performs a recursive decomposition of the Euclidean space into equal size quadrants until the minimum resolution is reached. Each node in the quadtree maintains a "frame" containing all relevant terrain attribute information for the region represented by that node. Thus, the frame-based quadtree provides a hierarchically-organized, spatial representation of terrain features. This approach also promotes efficient access to terrain feature information through a spatial-indexing technique made possible by the quadtree data structure.

The high-level *symbolic* representations facilitate reasoning about terrain information other than the basic gridded data points. The terrain knowledge, explicit in the representation, can be of great importance in solving path-planning problems, and predicting off-road vehicle performance.

B. MOBILITY MODELS

The formal study of mobility problems originated during the Second World War in response to a growing number of military vehicles that failed to negotiate various types of soft soil and mud in both the European and Pacific theaters of operation. Research continued at a low level until similar failures occurred during the Korean Conflict in the early 1950's. At that time, efforts intensified to analyze the relationship between both tracked and wheeled vehicles and the terrain on which the vehicles traveled. A coordinated program involving both the military and civilian sectors has provided a wealth of mobility information during the last four decades in the specific area of off-road (cross-country) vehicle performance. A detailed and comprehensive study, commissioned by the U. S. Army in 1969, examined the current state of the art for ground mobility models. The final report, published in 1971, entitled *An Analysis of Ground Mobility Models (ANAMOB)*, provides an excellent synopsis of off-road vehicle performance-evaluation techniques [Ref. 23].

In general, off-road performance for mobile robots involves the interaction of two key components: the vehicle and the terrain. Since the number of parameters associated with each of these components is large, it is difficult to model the complex relationships that exist among them. Therefore, the models that

have evolved focus on the interactions of single terrain features with the vehicle. As stated previously, a terrain feature refers to characteristics of the terrain such as soil type, slope, vegetation covering, and obstacles. The models, described in [Ref. 23] as *single-feature models*, measure cross-country vehicle performance in many critical areas, and can be divided into two fundamental categories: (1) soil-vehicle models, and (2) obstacle-vehicle models. Selected models from the obstacle-vehicle class, i.e., models specifically concerned with *soil-slope* relationships, are examined in detail. Evidently, these models have the most relevance to the minimum-energy path-planning problem.

To fully understand the models, several terms relating to soil properties need to be defined. The fundamental measure of soil strength, developed by the U.S. Army Engineer Waterways Experiment Station (WES), is defined in [Ref. 24] as the *cone index CI*. The "cone index" can be derived empirically by measuring the penetration depth of a cone-shaped instrument into various types of soil. The index predicts soil trafficability independent of vehicle speed and results in "go" or "no-go" assessment. To measure the effects of soil strength degradation due to multi-pass vehicular traffic, the soil is compacted within a cylinder and hammered to compress or remold it. The cone index is measured for the remolded soil and the ratio of the original cone index to the remolded cone index is defined by Bekker [Ref. 25] as the *remolding index RI*. The overall soil strength is defined as the *rating cone index RCI* expressed quantitatively as

$$RCI = CI \times RI. \tag{2.1}$$

The rating cone index can be described in terms of the number of passes a vehicles makes across a designated patch of soil. This index is defined in [Ref. 23] as the *vehicle cone index VCI*. A subscript attached to the vehicle cone index indicates the specific number of vehicle traversals. Thus, VCI_k indicates the soil strength required for k passes of a particular vehicle across a certain soil type. The values for the rating and vehicle cone indices are important in the analysis of the integrated mobility models discussed in Section II.B.2.

The nature of the off-road environment is such that slopes of varying degrees may be encountered in routine path traversals. The magnitude of the slope can have a significant impact on vehicle performance. At the extreme end of the spectrum, an unfavorable soil-slope combination can literally impede vehicle motion altogether. In addition to tractive failure, there is also the possibility of catastrophic overturn if the vehicle attempts to negotiate paths other than directly up or down the slope. The slope problem is, perhaps, the most significant area in determining off-road vehicle performance next to the soft-soil

problem [Ref. 23]. Despite this importance, there have been relatively few *slope models* developed to predict vehicle behavior. The slope models involve the effects of gravitational forces on the vehicle as it negotiates a particular terrain surface.

1. Slope Models

There are three principal slope models available for modelling vehicle motion on inclined terrain surfaces [Ref. 23]. The first model uses the concept of available tractive force or *drawbar pull* *DBP* to predict the slope-climbing capability of the vehicle. Drawbar pull is defined formally by Bekker [Ref. 25] as the difference between the gross tractive force and the motion resistance created from the soil and slope properties. The problem considers the vehicle on a slope as a static friction problem and defines a *tractive coefficient* equivalent to the ratio of the drawbar pull DBP_L measured on a level surface, to the vehicle weight W. The tractive coefficient can be interpreted as a coefficient of static friction. Generalizing the concept of drawbar pull on a level terrain surface, a sloped version is defined. The available tractive force on a sloped terrain surface DBP_S is expressed quantitatively in the ANAMOB Study [Ref. 23] as

$$DBP_S = \left[\frac{DBP_L}{W}\right] W\cos\theta - W\sin\theta, \tag{2.2}$$

where θ represents the slope of the terrain surface. The maximum negotiable slope is obtained when DBP_S is equal to zero. Thus, solving for θ, Eq. (2.2) can be rewritten as

$$\theta = \tan^{-1}\left[\frac{DBP_L}{W}\right]. \tag{2.3}$$

The U. S. Army Waterways Experiment Station (WES) attempted to validate the slope model described above using both wheeled and tracked vehicles. The tests occurred on sloped terrain surfaces up to twenty percent comprised principally of fine-grained soil. The results in comparing actual drawbar pull measurements with the predicted results of the model were accurate within one percent [Ref. 26].

The second model, proposed by the Land Locomotion Division (LLD), U. S. Army Tank and Automotive Command, addresses the problem of weight transfer in the vehicle as it negotiates a sloped terrain surface and the resulting impact on the net tractive force available. The basic premise of the model is that the front-to-rear shift in weight that occurs in a vehicle as it traverses an uphill slope increases the ground pressure at a certain point where the wheels or tracks meet the terrain. The increased pressure can be a source of tractive failure and, therefore, should be accounted for in predicting soil-slope trafficability.

The contact pressure is computed at both the front and rear of the vehicle taking into consideration both the slope of the terrain surface and the vehicle center of gravity. The contact pressure values are subsequently used to compute the front and rear sinkage and, thus, the resistance due to soil compaction and slope. The weight-transfer model has not been tested widely and, therefore, its use in cross-country vehicle performance evaluation has been limited.

The third model of interest focuses on the slope problem from the standpoint of vehicle stability. It attempts to predict vehicle performance when traversing a "contour" path, or a path that is perpendicular to the gradient, or maximum slope line. The model is simple in that vehicle stability is computed as a function of the height of the vehicle center of gravity and the distance between tires or tracks. The vehicle path traversals in the side-slope model are restricted to 90 degrees with the maximum slope line and, therefore, do not permit performance evaluation on any other permissible vehicle headings between the gradient and contour lines. Performance specifications for military vehicles routinely provide maximum side-slope information as part of an overall mobility assessment.

2. Integrated Models

There are two widely-used analytical mobility models that are designed to evaluate vehicle performance: (1) the Defense Mapping Agency/Engineering Topographic Laboratory (DMA/ETL) Cross-Country Mobility Model and (2) the U. S. Army Mobility Model (AMM). The Army Mobility Model is recognized as the *de facto* standard vehicle mobility prediction model [Ref. 11]. The primary consideration in the Army Mobility Model is the average speed a vehicle can sustain in traversing a path from a start point to a goal point on natural terrain. The average speed is computed as a function of the total area under evaluation that is permissible for travel. For example, a mobility prediction from the model may indicate that a particular type of vehicle can sustain a speed of n miles per hour within a given homogeneous region if it avoids the most difficult x percent of the terrain surface. Evidently, the average speed increases as the percentage of difficult areas it must avoid decreases.

Other mobility models incorporate "safety" factors in predicting the best routes of travel. Rowe and Lewis [Ref. 27] use the notion of detectability from hostile observers as a cost parameter in a three-dimensional search problem. Kanayama [Ref. 28] presents a mathematical theory of safe path planning that finds a locally minimum-cost path within a given equivalence class of paths. A safety index is used to select an appropriate cost function with consideration given to path safety and path length. As path safety is increased, longer paths are generated, and as path safety is decreased, shorter paths are obtained.

C. SPATIAL REASONING METHODS

Spatial reasoning is a broad research area that has many constituent subfields. A subfield of particular interest is that of *path planning*. There are a myriad of techniques available for finding optimal paths for both two-dimensional and three-dimensional problems. In general, path-planning problems can be divided into three fundamental categories: (1) obstacle-avoidance problems, (2) discrete geodesic problems, and (3) weighted-region problems. Since the minimum-energy path-planning problem addressed in this dissertation is a hybrid between the two-dimensional and three-dimensional problems, each of the above areas is reviewed for completeness.

There are several issues of importance in the obstacle-avoidance, discrete geodesic, and weighted-region path-planning problems. The first issue of concern is the *representation* of the terrain. As mentioned previously, the standard approaches rely on either a uniform grid of data points or the representation of homogeneous regions by convex polygons. The second issue involves the selection and implementation of an appropriate *search strategy* for the path-planning problem. Generally, the search techniques can be described as either exhaustive or heuristic, depending on the knowledge used to "direct" the search to the goal. The final issue of relevance to the path-planning problem is the selection of the optimality criterion and the corresponding development of an appropriate *cost function*. The cost function serves as an integral part of the search algorithm and can vary in the degree of complexity. A review of a particular path-planning approach will consist of an analysis of (1) the terrain representation selected, (2) the search strategy employed, and (3) the associated cost function used.

1. Obstacle-avoidance Problems

Obstacle-avoidance problems assume a binary partitioning of the cartographic map plane into entities traversable by the vehicle and entities considered obstacles. Several terrain representations are possible. The two-dimensional map plane can be represented by a uniform grid of data points, that is, a regular tessellation of the plane. The data points can also be represented in a hierarchical manner using the quadtree approach of Samet [Ref. 21]. The quadtree is a data compression technique that employs a recursive decomposition of the Euclidean space into equal size quadrants until a minimum resolution is obtained. An alternative terrain representation consists of a set of convex polygons superimposed on a background region, such that each polygonal region represents an obstacle region or a region that is impossible for the vehicle to traverse. The obstacle regions are usually disjoint.

In the grid-based approach, each data point is classified as either "go" or "no-go", typically by spatial averaging, and the obstacle-avoidance problem is relatively straightforward. The graph of gridded points can be searched using an *uninformed* technique, such as the Dijkstra algorithm [Ref. 29] (also known as the wavefront-propagation-search method), or an *informed* technique, along the lines of the A^* heuristic search algorithm [Ref. 5]. In both cases, the search of the obstacle space generates minimum-distance paths between pre-defined start and goal points. Regardless of which search technique is used to solve the problem, the inherent problems associated with the basic grid structure cannot be ignored.

For obstacles represented as convex polygons, the optimal path between a start point and a goal point is either a straight line between the two points (if the line does not intersect any obstacles) or a set of straight-line segments, each of which is constrained to pass through a vertex belonging to some obstacle region. In this case, the basic approach to solving the binary-case path-planning problem involves constructing a visibility graph (VGRAPH) of the obstacle space [Ref. 30]. The nodes of the visibility graph are the obstacle vertices and the links represent path segments connecting pairs of vertices such that each individual segment does not intersect any obstacle region. The visibility graph can be searched using any appropriate search technique (informed or uninformed) using distance as the criteria for optimization. The dominant cost in the VGRAPH approach is constructing the obstacle map.

An alternative method for finding the shortest path partitions the two-dimensional plane into regions. Each region is the locus of all goal points whose shortest path from the start point traverses the same sequence of nodes (in this case obstacle vertices). The search becomes relatively simple. It requires locating the goal point and then traversing the links in reverse order back to the start point to obtain the optimal path.

The *potential fields* approach to the shortest path problem described in [Ref. 31] models the goal as an attractive force and the obstacles as repulsive forces. The forces are strictly a function of distance from visible obstacles, i.e., clearance. Viewing the vehicle as a point, a repulsive function is computed at each data point in the grid, based on the known distance to visible obstacles. The optimal path is determined by finding a sequence of grid points from start to goal, minimizing a cost function that is a weighted sum of distance and repulsion. This method appears to be well-suited for local obstacle avoidance but can experience difficulties in long-range route planning. The path may lead to dead ends necessitating the use of backtracking operations. Thus, the potential fields approach seems to work best in conjunction with a global path-planning approach.

2. Discrete Geodesic Problems

The two-dimensional, obstacle-avoidance problem can be generalized to three dimensions by finding the shortest path for a vehicle constrained to move along a nonplanar surface. This problem is described by Mitchell [Ref. 32] as the *discrete geodesic problem.* As in the obstacle-avoidance problem, the terrain can be discretized and represented in various ways. The most common approach is a polyhedral representation with a surface of planar faces, edges, and vertices. Another is a grid-based representation where each cell in the grid is an elevation data point as mentioned in Section II.A.

The shortest path on a surface using the gridded representation can be found with any standard search algorithm, either informed or uniformed. Two common strategies employ the A^* algorithm or the uniform cost (Dijkstra) algorithm with costs computed as the three-dimensional Euclidean distance between either the four or eight nearest neighbors. The polyhedral representation is somewhat more complicated and finding the shortest path has been determined to be an NP-hard problem [Ref. 33]. Conceptually, the solution to the problem is accomplished by "unfolding" the polyhedron and flattening the surface so the start and goal can be connected by a straight line that remains within the flattened surface. The optimal path will never pass through the same region more than once as is evident in the above approach. There are, however, an exponential number of possible ways to unfold the polyhedron in order to produce the shortest path [Ref. 34].

3. Weighted-region Problems

The two-dimensional, binary-case, obstacle-avoidance problem can be generalized to allow different costs within regions. The cost per unit distance of travel defines a "weight" for the region and can range from one to $+\infty$. Finding the shortest path through a set of variable cost regions is described by Mitchell and Papadimitriou [Ref. 35] as the *weighted-region problem.* It is evident that the binary-case obstacle-avoidance problem is a special case of the weighted-region problem where the weights are constrained to be either one or $+\infty$, the former representing areas of free space for path traversals and the latter denoting obstacle regions. The weighted-region problem more closely represents the complex nature of the real world with the various types of soil, vegetation, lakes, rivers, and road networks, each having a different traversal cost associated with movement across it. As in the binary case problem, there are two possible representations that can be employed to describe the terrain: (1) a uniform grid of "weighted" data points or (2) a polygonal subdivision of "weighted" regions.

For the grid-based approach, the weight of a particular data point can be assigned by the properties associated with the point. A *feature vector* describing the relevant properties can be used to accomplish this task. Movement on the grid is based on the nearest neighbor connectivity assumption with costs assigned according to a pre-determined formula. The cost can be computed as the average of the two grid-point costs or weighted differently using some other heuristic. Any of the previously discussed search techniques is appropriate for finding the least-cost path between a start point and goal point on the weighted grid. A^* and uniform cost search are two of the most common strategies. The advantage of a more precise representation of region costs can be offset by the disadvantage of digitization bias as discussed previously. In addition, for maps with very few features or large, homogeneous areas, the entire array of weighted data points must be included and must participate in the search process. This results in the needless expansion of many data points. Richbourg [Ref. 6] has examined the grid-based weighted-region approach using the dynamic programming paradigm for searching the graph (also termed the wavefront-propagation method) and confirmed the above observations.

An alternative approach exploiting Fermat's Principle of optics has been explored independently by Mitchell and Papadimitriou [Ref. 35], Richbourg [Ref. 6], and Rowe [Ref. 36]. This approach relies on a homogeneous regions model that partitions the terrain into polygonal regions of uniform traversability cost. The local path behavior at region boundaries can be determined by applying Snell's Law of Refraction at each crossing point. The path behavior models that of a light ray at it travels through various types of media. Given this representation guideline for path behavior, the problem is to find the shortest path (using the weighted Euclidean metric) from a start point to a goal point.

The weighted-region problem is solved by Mitchell and Papadimitriou [Ref. 35] by triangulating each homogeneous cost region and then employing a dynamic programming search paradigm (Continuous Dijkstra Algorithm). The algorithm uses Snell's Law to generate non-overlapping "intervals of optimality" on the boundaries of the triangles designated as "wedges". The wedges represent the least-cost (in the weighted sense) path from the start point to that boundary. Within a given wedge, the minimum-cost path is computed using Snell's Law and then substituted for the known cost (moving from node to node) required for the cost function in the search algorithm. Thus, the Snell's Law cost is used primarily to identify minimum-cost wedges from the start point to the most recent boundary reached in the interval of optimality. The algorithm continues until every least-cost wedge for each interval of optimality for each boundary in the triangulated map has been created and stored. Thus, given the start and goal points within

the set of homogeneous cost regions, the least-cost path can be found by selecting the appropriate wedge and then solving Snell's Law iteratively.

A more informed strategy, attributed to Richbourg [Ref. 6], uses A^* search together with a set of heuristics and pruning criteria to improve the average case performance of the Snell's-Law-directed solution to the weighted-region problem. The search does not require triangulated terrain. Given an initial start point and goal point, the algorithm begins by partitioning the map into two initial "wedges", one of which contains the goal. A feasible path to the goal is obtained by ignoring cost regions and considering only obstacle regions. The feasible path to the goal is an upper bound on the optimal path solution and can be used to construct a "limiting" ellipse to reduce the size of the search space. Using a lower-bound evaluation, a refinement operator creates "sub-wedges" based on the Snell's Law path to the closest unsolved search point within the wedge: that is, a branching factor of three (the path plus two adjacent new wedges from the split). A comprehensive methodology is developed to evaluate upper and lower cost bounds on start-to-goal paths through "wedges". The wedge having the best lower-bound cost is the first to be refined at every step. This implies an ordering of the wedges according to the likelihood of containing the optimal path. The new search state consists of a wedge, the least-cost path from start found thus far in the search, and a lower-bound evaluation for the wedge to the goal. The search terminates when the cost of the best path found is less than the lower-bound evaluation for every state in the search space. The search can also terminate if the lowest lower-bound wedge cost exceeds the upper-bound feasible cost.

4. Some Approaches to the Optimal-path-planning Problem

Most path-planning and obstacle-avoidance problems assume an isotropic medium for the search space. Gaw and Meystel [Ref. 4], in the minimum-time navigation problem, propose a 2-1/2 dimensional terrain representation using "isolines" or contour lines indicating uniform elevations. In contrast to the uniform discretization of the grid-based approach, the isolines are discretized at an arbitrary resolution forming contour lines consisting of a finite set of line segments. This process polygonalizes the isolines. The advantage to this representation is that it avoids the wasteful approach of uniform discretization and provides for a more flexible map structure. A significant disadvantage is the loss of key terrain information that occurs with any contour-line representation. Specifically, the tops of hills and the bottoms of valleys can be distorted. There is also error introduced in the polygonalization of the isolines, although not directly considered in the model. The vertices of the polygonalized isolines are the search points on the map, and navigation is accomplished by moving from vertex to vertex. Obstacles are

represented directly by polygons that are superimposed over the discretized isolines. The search algorithm used is A^* with the standard cost function $f = g + h$, such that g is the cost from the start point to the current location, and h is the heuristic evaluation of the estimated cost from the current location to the goal. The evaluation function employed is computed for a straight line from the current node to the goal, accounting for differences in elevations. The cost function is more complex and relies on the physical properties of the vehicle and its behavior on a sloped surface. In the 2-1/2 model, the acceleration and braking of the vehicle are considered in developing the anisotropic cost function replacing the standard Euclidean distance as the estimate of the time required to traverse a particular path segment. The physical model assumes maximum power output by a vehicle and employs energy equations of motion in computing traversal time. A three-valued cost function is proposed for traversing across the isolines. The "time-cost" function for any path segment can be computed using the value for the slope of the segment; that is, there is a unique formula for "uphill", "downhill" and "level" path segments. Cost penalties are imposed heuristically for path segments with endpoints on the same isolines based on how close the segment comes to an adjacent isoline. The resulting paths produced by the search algorithm avoid straight-line trajectories if those trajectories would necessitate traveling up and down slopes.

Another anisotropic approach to route planning was proposed by Parodi [Ref. 8]. The route planning system for an autonomous vehicle employs a two-level global map representation: a grid level and a symbolic level, as in the Defense Mapping Agency databases discussed in Section II.A. The symbolic level is a compressed data representation describing terrain features by a set of polygons. The grid, or "pixel" level is used for the assignment of cost factors and conducting the actual search. The path-planning model depends on the vehicle properties, and a separate model is used to describe vehicle behavior with respect to the terrain. The model addresses vehicle stability (roll over conditions), energy consumption, detectability, usage (mean time before failure), and maximum speed. These factors are dependent on the configuration of the terrain. Energy consumption is represented by a bivariate polynomial taking into account both the speed of the vehicle and the slope of the terrain. A global state description is employed that records (x,y) position, average speed, average heading, and average stability (roll and pitch). The anisotropic cost to go from one grid point to one of its eight nearest neighbors is a linear combination of weighted elementary costs for each of the criteria to be optimized. The graph search involves a combination of dynamic programmimg (as in wavefront propagation) and relaxation that facilitates backward searching: that is, retracing a previous route, turning around, and moving forward again. The algorithm is able to develop optimal paths that avoid obstacles, take advantage of high-speed corridors such

as roads, and move off-road into traversable forest areas when probability of detection is high. The paths can move away from the goal at times in order to satisfy the cost-minimization criteria or to avoid obstacles.

Long-range, strategic path-planning through variable terrain data was explored by Mitchell and Kiersey [Ref. 37]. The BITPATH long-range planner for an autonomous vehicle operates on a grid of data points from a Defense Mapping Agency database. The 64-bit array of data points contains elevation and cultural (feature) data at 12.5 meter resolution. The planner uses the data to develop a composite cost of movement between adjacent grid points, that is, the eight nearest neighbors. The cost function accounts for differences in elevations of data points and has independent components for movement costs on roads and across natural terrain. The movement costs are the reciprocal of the maximum speed either on or off road. The movement cost between any two adjacent grid points forms an arc cost between neighbors. The minimum-cost path between a given start point and goal point is found by using the Dijkstra algorithm, or the dynamic programming search technique. The algorithm is uninformed and appears to expand nodes without a sense of direction; there is no heuristic evaluation that assists in guiding the search toward the goal. The problem of digitization bias also exists in every grid-based terrain representation. An alternative version of BITPATH employs an informed search algorithm (A^*) with three heuristic evaluation functions from which to choose as well as a "heuristic level". The heuristic level serves as a "weighting factor" for the three functions. The first heuristic evaluation function uses the Euclidean distance to the goal. The second function truncates the floating point value computed in the first function, and the last estimate to the goal is a function of the sum of the (absolute value) differences in the x and y distances to the goal. By varying the weighting factor, paths can be obtained that are less than optimal but that run significantly faster.

The path-planning system developed by Linden, Marsh, and Dove [Ref. 38], for the Autonomous Land Vehicle (ALV) uses a uniform grid of data points for the terrain representation such that each data point indicates the traversability factor for a vehicle at that point. A cost matrix is developed to estimate the probable costs of traversal between adjacent grid neighbors using any available information. Routes are generated using the search strategy of dynamic programming. The best path to the goal is found by optimizing some *figure-of-merit (FOM)* which represents the total cost to reach a particular grid point from the start point. The algorithm begins by assigning an initial FOM of infinity to all grid points except the goal, which has a FOM of zero. The algorithm iterates over the entire grid, and at each point replaces

the current FOM with the sum of the FOM at the neighboring point plus the cost to traverse the link between the two points ($FOM_{NEW} = FOM_{OLD} + COST_{ARC}$), if the cost is lower. Otherwise, the previous FOM is retained. The iteration continues until a "steady state" is reached and it is no longer possible to improve the FOM at any point on the grid. The control strategy for the iteration allows the expansions to "sweep" out across the grid in a left-to-right or top-to-bottom manner checking only grid points that have been examined during the current sweep. The least cost path can be determined from the computed FOM grid by following the maximum gradient of the FOM's in decreasing order until the goal is reached.

An approach to partitioning the terrain for path-planning problems is described by Kwan, Zamiska and Brooks [Ref. 19] and involves a mixed representation of free space. Concave obstacles are divided into connecting convex obstacles and the remaining free space is split into "channels" and "passage regions". Channels are the narrow free space between obstacle regions and are represented by generalized cones. Passage regions are larger areas of free space that are represented by convex polygons. Spatial relationships are maintained between the obstacle regions that help identify the critical channel and passage regions. A search graph of collision-free path segments is created using critical points from the channel and passage regions. An A^* search algorithm finds the shortest path from the given start and goal points. The cost function employed for the minimization is simply the Euclidean distance.

Path relaxation is a combined search technique that uses elements from the grid-based methods and the potential-fields methods [Ref. 7]. The global grid-based search finds an approximate path to goal and then the path is modified locally through a relaxation process to reduce the overall traversal cost. After an initial grid size is selected, the costs are assigned to paths on the grid and then a graph search finds the best path from the start to the goal. The best path must satisfy conflicting requirements: shorter path length, greater distance away from obstacles, and less distance in unknown areas. After the cost is computed for each node in the grid, links connecting the eight nearest neighbors are established. An A^* search finds the least-cost path from the start point to the goal point. The search algorithm uses a Euclidean distance metric in its evaluation function and therefore, finds the minimum-cost path to the goal. Grid optimality may not be a good measure of the true path optimality because of digitization bias and other anomalies that occur due to the cost function. A path-relaxation phase optimizes the position of the grid point on the path to minimize the total cost. This is accomplished by perturbing the grid points in turn and adjusting the costs based only on local information. The object is to minimize the cost of the path sections on either side of the grid point being moved.

D. SUMMARY

A review of three significant areas relevant to the minimum-energy path-planning problem has been presented. Terrain representations, mobility models, and search techniques all play an important part in the path-planning solution developed in this dissertation. In addition to the fundamental terrain representations and mobility models, a wide cross section of search techniques is presented, both for two-dimensional and three-dimensional path planning. Perhaps the most important consideration is the cost function associated with the search strategy. This area has particular importance for the mathematical model of vehicle-terrain interaction presented in the next chapter and is a critical element of the optimal-path-planning algorithm that generates the minimum-energy path within a specified set of stability and motion constraints.

III. MATHEMATICAL MODEL OF VEHICLE-TERRAIN INTERACTION

A. INTRODUCTION

A key component to solving the minimum-energy path-planning problem is a sound mathematical model that adequately reflects the characteristics of vehicle performance in an off-road environment. It is necessary to develop a model that facilitates the abstraction of essential information for the objectives of path planning while ignoring other, less relevant information. With that goal in mind, the many complexities of vehicle-terrain interaction can be simplified by considering vehicle motion from an *external* rather than an *internal* perspective. The *towed vehicle model* posits vehicle motion resulting from a towing force applied to a hypothetical cable attached to the front of the chassis. The tension force pulling on the cable must be sufficient to overcome all resistive forces and keep the vehicle moving at low, constant speed. The alternative approach considers motion resulting from the propulsive forces generated by the internal combustion engine of the vehicle.

The primary focus of the towed vehicle model is on the forces of resistance resulting from the operation of the internal mechanical systems of the vehicle and motion of the vehicle over a wide range of natural terrain surface configurations and surface compositions. Thus, the model contains components that are strictly *vehicle dependent*, components that are strictly *terrain dependent*, and components representing a hybrid of *vehicle-terrain dependencies*. There are many ways in which the motion of a vehicle can be described. The towed vehicle model considers vehicle motion at three, increasingly-complex levels of abstraction: (1) simple plane motion, (2) cylindrical surface motion, and (3) generalized three-dimensional motion. In all cases, the model relies on a direct application of Newton's First Law of Motion and the general principles of work and energy. Throughout the development of the mathematical model, certain fundamental assumptions are made that are central to the solution of the global, off-road path-planning problem.

Assumption 3.1: The vehicle is treated as a particle or point mass.

The first assumption is a simplification permitted because the model ignores rotational kinetic energy. A vehicle can still have massless extensions, but has negligible moment of inertia about its center of gravity.

Assumption 3.2: The vehicle moves between any two specified points in a straight line at low, constant speed.

The second assumption is justified due to the nature of off-road travel over arbitrarily-sloped terrain. Vehicles must necessarily move at speeds that insure safety over negotiable terrain. As a result, the average speed of the vehicle is considerably lower than for on-road travel.

B. SIMPLE PLANE MOTION

The towed vehicle model is first described in terms of simple *plane motion*. If a vehicle is confined to moving along a specified path, its motion is *constrained*. From Assumption 3.2, the vehicle is constrained to move along a fixed, straight-line path, and therefore, has one degree of freedom in its motion. This type of motion is described in [Ref. 39] as *rectilinear motion*. A two-dimensional, Cartesian coordinate system is established with the x-axis representing the axis tangential to motion and the y-axis representing the axis normal to motion.

1. Forces and Equations of Motion

One approach to analysis of motion involves the definition of forces acting on the vehicle and the creation of a *free-body diagram*. The free-body diagram isolates the vehicle from all contacting or influencing bodies and substitutes those bodies by the appropriate forces exerted on the vehicle [Ref. 39]. For the towed vehicle model, there are four principal forces of interest: (1) the towing force $\vec{F}_{TOW}$, (2) the force of gravity $m\vec{g}$, (3) the normal force $\vec{N}$, and (4) the force of friction $\vec{f}$. The towing force $\vec{F}_{TOW}$ is defined as the tension force in a towing cable necessary to keep a vehicle moving at constant speed along its path. The force of friction $\vec{f}$ represents a resistive force that always acts in the opposite direction to the applied towing force $\vec{F}_{TOW}$. The normal force $\vec{N}$ opposes the normal component of the gravitational force. The velocity vector $\vec{v}$ indicates the direction of vehicle motion in the plane and is parallel to the tangential motion axis. Once all pertinent forces have been identified, the appropriate energy equations can be derived. Figure 3.1 illustrates a complete free-body diagram for a vehicle in plane motion.

The first fundamental concept applicable to the development of the towed vehicle model is Newton's First Law of Motion which is stated in [Ref. 40] as follows:

- *If the resultant force acting on a particle is zero, the particle will remain at rest (if originally at rest) or will move with constant speed in a straight line (if originally in motion).*

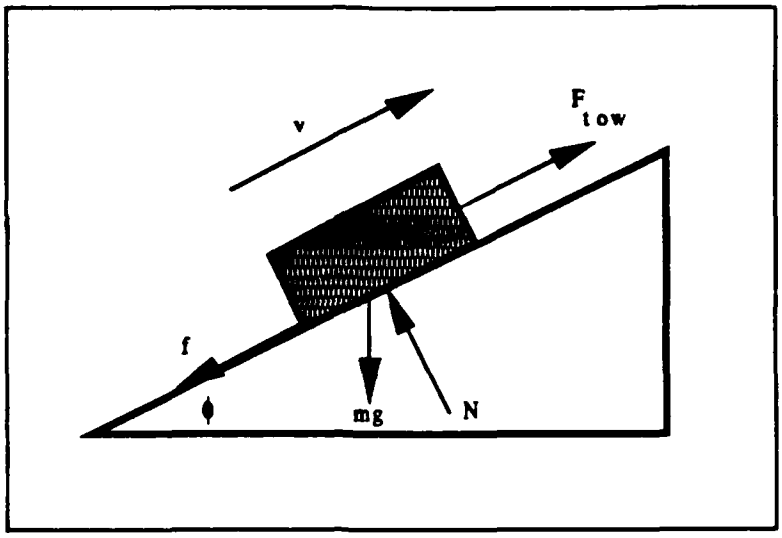

Figure 3.1 Free-body Diagram

From Assumption 3.2, it is evident that the model posits a system that is in a state of *static equilibrium* in which the equation of motion is expressed as

$$\sum \vec{F} = 0. \tag{3.1}$$

The resulting scalar components of the equation of motion in the normal and tangential directions become

$$\sum F_x = 0 \tag{3.2a}$$

and

$$\sum F_y = 0. \tag{3.2b}$$

From the free-body diagram, the component forces are expressed as

$$\sum F_x = F_{TOW} - f - mg \sin\phi = 0 \tag{3.3a}$$

and

$$\sum F_y = N - mg \cos\phi = 0 \tag{3.3b}$$

where $mg\cos\phi$ and $mg\sin\phi$ represent the normal and tangential components of the gravitational force.

Thus, from Eq. (3.3a), the total force required to keep the vehicle moving at constant speed is

$$F_{TOW} = f + mg\sin\phi. \tag{3.4}$$

Specifically, Eq. (3.4) represents the tension force necessary to overcome the forces of friction and gravity and to pull the vehicle at a constant speed across a surface.

2. Work and Energy

The second fundamental concept used in the towed vehicle model is that of *work*. Work is defined as the cumulative effect of a force $\vec{F}$ over a differential displacement $d\vec{s}$ at the point where the force is applied, and is expressed quantitatively in [Ref. 39] as

$$dU = \vec{F} \cdot d\vec{s}. \tag{3.5}$$

The work done by a force is defined as *energy*. Using the definition of Eq. (3.5), the total energy expenditure or work done by the towing force $\vec{F}_{TOW}$ during a displacement $d\vec{s}$ from position s_1 to s_2 is equivalent to

$$U_{s_1 \to s_2} = \int_{s_1}^{s_2} \vec{F}_{TOW} \cdot d\vec{s}. \tag{3.6}$$

The magnitude of the dot product of the force and displacement vectors can be expressed as

$$\vec{F}_{TOW} \cdot d\vec{s} = F_{TOW} \cdot ds\cos\theta \tag{3.7}$$

where θ represents the angle between $\vec{F}_{TOW}$ and $d\vec{s}$. From the free-body diagram, it is evident that the towing force $\vec{F}_{TOW}$ acting on the vehicle is in the direction of the displacement and thus, Eq. (3.6) becomes simply

$$U_{s_1 \to s_2} = \int_{s_1}^{s_2} F_{TOW} \cdot ds. \tag{3.8}$$

If the towing force F_{TOW} is held constant during the displacement ds, Eq. (3.8) can be rewritten as

$$U_{s_1 \to s_2} = F_{TOW} \int_{s_1}^{s_2} ds \tag{3.9}$$

where s is the *distance* defined by $s_2 - s_1$. With this definition, it follows that

$$U_{s_1 \to s_2} = F_{TOW} \cdot s. \tag{3.10}$$

Substituting the equivalent opposing forces, Eq. (3.10) becomes

$$U_{s_1 \to s_2} = (f + mg \sin\phi)s. \tag{3.11}$$

The resulting expression for the total work done by the towing force $\vec{F}_{TOW}$ over a finite distance s is defined in the *basic energy equation* as

$$U_{s_1 \to s_2} = fs + mg \sin\phi s \tag{3.12}$$

where fs is the work done against the *friction forces* and $mg \sin\phi s$ represents the work done against the *gravitational force* component. From the above analysis, it is observed that only the tangential components of the forces acting on the vehicle can do work. Eq. (3.12) can be interpreted as the total energy required to pull a vehicle at constant speed across an arbitrarily-sloped terrain surface for a specified distance.

The forces in Eq. (3.11) are classified in [Ref. 39] as *conservative forces* and *nonconservative forces*. When work is done against a nonconservative force such as friction, mechanical energy is dissipated and converted into heat energy. The negative work that results represents a net loss of mechanical energy. Work done against a conservative force such as gravity, is stored in the form of *potential energy*. To analyze the effects of potential energy, a different two-dimensional Cartesian coordinate system is established, with the x-axis representing an arbitrary, horizontal reference axis or *datum*, and the y-axis representing the vertical elevation of the terrain. Potential energy exists in a vehicle because of its position relative to a reference position or datum. The gravitational potential energy V_g of the vehicle is defined in [Ref. 39] as the work done against the conservative force of gravity to elevate the vehicle a distance h above the datum. If V_g at the datum is assumed to be zero, the potential energy at an arbitrary position above the datum is expressed quantitatively as

$$V_g = mgh \tag{3.13}$$

where $h = \Delta y$ in the coordinate system defined above. Thus, the total change in potential energy when the vehicle moves from one elevation at $h = h_0$ to another elevation at $h = h_1$ is

$$\Delta V_g = mg\,(h_1 - h_0) \tag{3.14}$$

or equivalently,

$$\Delta V_g = mg\,\Delta h. \tag{3.15}$$

The corresponding work done against the gravitational force by the vehicle is the negative of the potential

energy change in Eq. (3.15). As the vehicle returns to its original lower datum plane, the potential energy $mg\,\Delta h$ may be converted into energy of motion, called *kinetic energy*, or perform *mechanical work;* e.g., work against friction forces. Evidently, by Assumption 3.2, the kinetic energy is invariant during vehicle motion between any two points and is therefore, neglected during the global path-planning problem. This approach is supported by the fact that humans ignore kinetic energy in long-distance route planning and consider it in local planning only.

The computation of energy costs is related to path planning using the mathematical concept of a *vector.* A vector is defined, informally, as any entity that is specified by a magnitude and a direction [Ref. 41]. For the towed vehicle model, a vector describes the discrete path followed by a vehicle. All vectors are assumed to lie in the standard x-y coordinate plane. If point O defined by Cartesian coordinates (x_0, y_0) represents the origin of the coordinate system and initial position of the vehicle, and point P defined by coordinates (x_p, y_p) represents the final position of the vehicle, then the vector $\vec{s}$ describes a unique directed line segment $\overrightarrow{OP}$. If $\vec{s} = \overrightarrow{OP}$, then the coordinates of P are defined as *components* of $\vec{s}$ represented by the 2-tuple (s_1, s_2) and $\vec{s}$ is the *position vector* of point P. The components of vector $\vec{s}$ can be expressed in terms of *unit vectors* defined for each of the two coordinate axes; i.e., $\vec{i} = (1,0)$ and $\vec{j} = (0,1)$. Thus, $\vec{s}$ is given by $\vec{s} = s_1\vec{i} + s_2\vec{j}$. The term $s_1\vec{i}$ is the vector component of $\vec{s}$ along the x-axis and s_1 is the *scalar* component of $\vec{s}$ in the $\vec{i}$ direction. A similar interpretation holds for the term $s_2\vec{j}$ with the focus on the y-axis.

Definition 3.1: For simple plane motion, a two-dimensional vector $\vec{s} = (s_1, s_2)$ is defined as a *path segment* S where s_1 and s_2 represent the components of $\vec{s}$ along the x and y axes, respectively. The length or magnitude of path segment S, denoted by $|\vec{s}|$, is designated as the *path distance* d and is computed as the two-dimensional Euclidean distance defined as

$$d = (s_1^2 + s_2^2)^{1/2}. \tag{3.16}$$

With this definition, the basic energy equation, Eq. (3.12), can be rewritten in terms of path segment distance as

$$U_{s_1 \to s_2} = fd + mg\sin\phi d \tag{3.17}$$

Subsequent to defining the vehicle path by its position vector, the configuration of the path segment S can be described quantitatively by introducing the mathematical concept of *slope*, which represents the vertical change in the path elevation over a specified horizontal distance, and is defined as

$$slope = \frac{\Delta y}{\Delta x}. \tag{3.18}$$

Using Eq. (3.18), the slope of the path segment S can be written in terms of angular displacement from the datum, expressed as

$$\phi = \tan^{-1}\frac{\Delta y}{\Delta x}, \quad -90 < \phi < 90. \tag{3.19}$$

By geometry, it is evident that the change in elevation Δy over the path segment S is defined as

$$\Delta y = d\sin\phi. \tag{3.20}$$

The component of the position vector $\vec{s}$ along the x-axis is significant for path-planning applications, and is defined by the method of vector projection using the inner or dot product operation.

Definition 3.2: Let $\vec{i} = (1,0)$ be a unit vector along the x-axis, or datum and let $\vec{s} = (s_1, s_2)$ represent an arbitrary position vector defining a path segment S. The projection of $\vec{s}$ onto the x-axis or the vector component of $\vec{s}$ in the $\vec{i}$ direction $s_1\vec{i}$, is defined as a *projected path segment* PS. The length or magnitude of the projected path segment PS is designated as the *projected path distance* D, expressed quantitatively as

$$D = |\vec{s}|\cos\phi = \vec{s}\cdot\vec{i} \tag{3.21}$$

or equivalently,

$$D = d\cos\phi. \tag{3.22}$$

Figure 3.2 illustrates a path segment S and its projection PS in the Cartesian coordinate plane. Given the definition of path segment projection, Eq. (3.17) can be rewritten as

$$U_{s_1 \to s_2} = \frac{fD}{\cos\phi} + mgh. \tag{3.23}$$

where $h = \Delta y$. The two components of Eq. (3.23) represent the energy required to overcome the effects of friction and gravity, respectively, and tow a vehicle at constant speed across a path segment of a specified distance. Having examined the potential-energy component, a more detailed analysis of the resistive component follows.

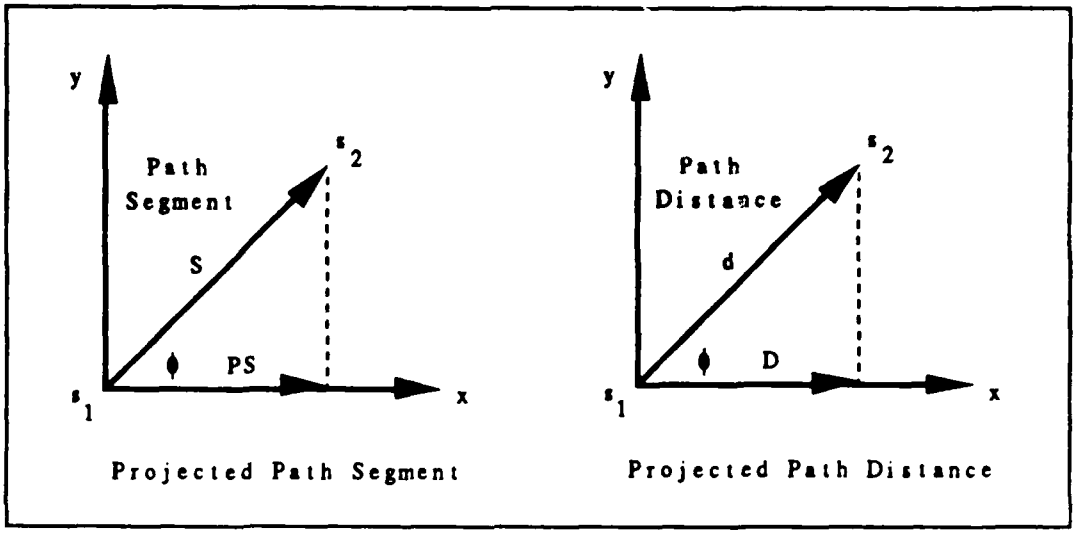

Figure 3.2 Single Path Segment and Projection

3. Motion Resistance

a. Resistive Forces

With regard to resistive forces, contacting surfaces can be classified as either smooth or rough. For perfectly smooth surfaces in contact with one another, it is assumed the only force exerted by one surface on the other is normal to the surface [Ref. 40]. In this situation, there is complete freedom of movement between the surfaces. In reality, however, a certain degree of surface roughness exists that serves to impede this free movement. When two surfaces exhibiting the roughness property are in contact, tangential forces can develop when the surfaces are moved against one another [Ref. 40]. The tangential forces, providing resistance to motion are called *friction forces*. It is evident from Eq. (3.23) that the principal resistive forces affecting vehicle motion are due to friction.

Friction is defined, generally, as the force distribution at a surface of contact between two objects that prevents or impedes sliding motion between the objects. Friction forces can be divided into two general categories: (1) fluid friction, and (2) dry or Coulomb friction [Ref. 40]. Fluid friction exists when a fluid or lubricant separates the sliding surfaces of two objects. The force necessary to initiate motion in this situation is the force required to shear the lubricant [Ref. 42]. In this dissertation, it is assumed that one source of fluid friction, the drag force on the vehicle due to air resistance, is negligible. This assumption is

justified by the fact that vehicles are forced to travel at relatively low speeds in off-road environments due to the complexity of natural terrain. Coulomb, or dry friction results from the forces of adhesion between the contact regions of the surfaces which are microscopically irregular. For sliding objects, it can be shown experimentally that the amount of Coulomb friction is nearly independent of the area of surface contact and that the friction force is proportional to the load or weight pressing the surfaces together [Ref. 40].

Within a vehicular system, there are many potential sources of Coulomb friction due to the large number of moving parts. If the parts are not lubricated or only partially lubricated, it can be assumed that there is direct contact between components [Ref. 40]. For example, journal bearings and thrust bearings are sources of axle friction and disk friction, respectively. This type of friction can be generally characterized as *bearing friction*. A significant portion of engine power is consumed overcoming the effects of bearing friction. The other contributing source of Coulomb friction is due to *rolling resistance*. Rolling resistance occurs when a wheel moves freely over a surface. The primary source of friction in rolling appears to be the dissipation of energy due to the deformation of the objects in contact [Ref. 42]. The existence of rolling resistance can be demonstrated, conceptually, by observing that a vehicle rolling along a level, frictionless surface will eventually come to a stop even in the absence of significant bearing friction. In this situation, the resistive force acting against the vehicle is attributed strictly to rolling resistance.

For the towed vehicle model, the friction forces can be described by a *hierarchy* of resistive forces. The hierarchy attempts to isolate the various sources of Coulomb friction by progressively accounting for the resistive forces at three distinct levels. The first level is concerned strictly with the friction forces resulting from the internal moving parts of a vehicle; i.e., bearing friction.

Definition 3.3: The resistive force associated with the internal moving parts of a vehicle is defined as *moving element friction* MEF, and is equivalent to the bearing friction and all other sources of friction resulting from the movement of those parts. The towing force F_{TOW} required to overcome friction and keep a hypothetical vehicle with rigid wheels moving at constant speed across rigid, level terrain is equal to the *moving element friction force* F_{MEF}, expressed as

$$F_{MEF} = F_{TOW}. \tag{3.24}$$

The intent of defining moving element friction is to isolate the forces of resistance strictly associated with the internal moving parts of the vehicle from the forces of resistance attributed to wheel or terrain surface deformation. The assumption of rigid wheels and a rigid, level surface implies an absence of either wheel

or terrain surface deformation and thus, eliminates any consideration of rolling resistance. Having established a baseline resistive force, the next level in the hierarchy can be developed.

Definition 3.4: The friction force associated with the deformation properties of the wheels or tracks of a vehicle in contact with a rigid, level surface is defined as *rolling element friction* REF. The difference between the towing force F_{TOW} on a rigid surface and the internal moving element friction force F_{MEF} represents the *rolling element friction force* F_{REF}, expressed quantitatively as

$$F_{REF} = F_{TOW} - F_{MEF}. \tag{3.25}$$

The assumption of vehicle contact with a hard surface implies the presence of wheel/track deformation and the absence of terrain surface deformation, thereby isolating the friction due to rolling resistance. Moving from hard surfaces to arbitrarily soft terrain surfaces, the third level of the resistance hierarchy, isolates the friction forces attributed to soil deformation.

Definition 3.5: The resistive force related to the deformation characteristics of the terrain surface is defined as the *soil deformation force* F_{SD} and represents the difference between the towing force F_{TOW} on a level, soft terrain surface and the combined resistances due to moving element friction MEF and rolling element friction REF. The soil deformation force F_{SD} can be written as

$$F_{SD} = F_{TOW} - F_{MEF} - F_{REF}. \tag{3.26}$$

Thus, all resistive forces defined for the towed vehicle model are assumed to be the result of various types of Coulomb friction.

Assumption 3.3: The resistive forces due to moving element friction, rolling element friction, and soil deformation are all proportional to the normal force N and are independent of vehicle velocity and terrain slope.

This assumption is of fundamental importance to the towed vehicle model and is generally supported by Coulomb's Laws and experimental evidence.

b. Energy Efficiency

A topic of paramount importance in the analysis of resistive forces for vehicular systems is the issue of *energy efficiency*. Energy efficiency, as related to resistive forces, is expressed quantitatively in [Ref. 43] by the introduction of a dimensionless parameter ε, called *specific resistance*, defined as

$$\varepsilon = \frac{U}{mgD} \tag{3.27}$$

where U is the total energy required to travel a distance D on level terrain, and mg is the weight of the vehicle in motion. To further expand the notion of specific resistance, it is useful to introduce the concept of *power* P or the time rate of doing work. The power developed by a force $\vec{F}$ which does an amount of work U is

$$P = \frac{dU}{dt} \tag{3.28}$$

or equivalently,

$$U = \int P \, dt. \tag{3.29}$$

Using the definition of differential displacement $ds = vdt$, Eq. (3.29) becomes

$$U = \int_{s_1}^{s_2} \frac{P}{v} ds. \tag{3.30}$$

The resulting expression for the total energy is

$$U = \frac{PS}{v} \tag{3.31}$$

where S is the distance traveled. For constant speed on level terrain, the ratio of the power required to tow the vehicle to the product of the vehicle weight and speed is defined in [Ref. 44] as *mechanical specific resistance* ε, expressed quantitatively as

$$\varepsilon = \frac{P}{mgv}. \tag{3.32}$$

Using an alternative definition of power $P=Fv$ derived from Eqs. (3.5) and (3.28), and assuming F represents the towing force F_{TOW}, the definition of Eq. (3.32) can be rewritten as

$$\varepsilon = \frac{F_{TOW}}{mg}, \tag{3.33}$$

or simply,

$$F_{TOW} = \varepsilon mg \tag{3.34}$$

Eq. (3.34) can be interpreted as the force required to overcome all resistive forces and pull the vehicle across a level terrain surface at constant speed. Thus, the term εmg represents a *composite resistive force* opposing the towing force in the state of static equilibrium. On level terrain, it is assumed that the resistive forces are the result of various types of friction, eliminating the force of gravity from consideration. The mechanical specific resistance ε is a composite resistance that includes components attributed to internal moving element friction MEF, rolling element friction REF, and soil deformation SD. The following definition formalizes the concept of a composite resistive force for a vehicular system.

Definition 3.6: The total resistive force that must be overcome by the towing force F_{TOW} to insure that a vehicle keeps moving at constant speed is defined as the *motion resistance force* F_{MR}. The motion resistance force consists of an internal resistive force F_{INT} and an external resistive force F_{EXT} represented by the expression

$$F_{MR} = F_{INT} + F_{EXT}. \tag{3.35}$$

Referring to the hierarchical description of friction developed earlier, the internal and external components of resistive forces can be defined.

Definition 3.7: The internal resistive force F_{INT} represents the internal losses within the mechanical systems of a vehicle due to bearing friction and rolling element deformation friction. The resistive force is measured on *hard, level terrain* at constant speed and is defined as

$$F_{INT} = (\varepsilon_{MEF} + \varepsilon_{REF})mg \tag{3.36}$$

where ε_{MEF} represents the component of specific resistance related to internal moving element friction and ε_{REF} represents the component of specific resistance associated with rolling element friction.

From Definition 3.7, it is evident that the internal resistive force F_{INT} is strictly vehicle dependent and does not include additional friction effects resulting from the composition or configuration of the terrain surface. Having isolated the forces of internal resistance, the external component is described.

Definition 3.8: The external resistance force F_{EXT} represents the external losses due to soil deformation work. The resistance is measured on *level terrain* at constant speed without restricting surface hardness, and is expressed as

$$F_{EXT} = \varepsilon_{SD} mg \tag{3.37}$$

where ε_{SD} represents the component of specific resistance associated with soil deformation.

In contrast to the internal resistive force F_{INT} the external resistive force F_{EXT} is dependent on both the vehicle and the terrain. For motion over level surfaces, the external component of specific resistance ε_{SD} differs from zero only on soft soils.

Using the results of Eqs. (3.35), (3.36), and (3.37), it is evident that the motion resistance force F_{MR} opposing the towing force F_{TOW} is equal to the sum of the component resistive forces F_{MEF}, F_{REF}, and F_{SD}, expressed quantitatively as

$$F_{MR} = \varepsilon_{MEF} mg + \varepsilon_{REF} mg + \varepsilon_{SD} mg \tag{3.38}$$

or equivalently,

$$F_{MR} = (\varepsilon_{MEF} + \varepsilon_{REF} + \varepsilon_{SD}) mg. \tag{3.39}$$

From Definition 3.6 and Eq. (3.39), a unified specific resistance is developed that provides a measure of the total resistance to the towing force for a vehicle moving at constant speed.

Definition 3.9: The *total specific resistance* ε is the sum of the component specific resistances ε_{MEF}, ε_{REF}, and ε_{SD} and is written as

$$\varepsilon = \varepsilon_{MEF} + \varepsilon_{REF} + \varepsilon_{SD}. \tag{3.40}$$

Thus far, in the towed vehicle model, total specific resistance ε has been defined on level terrain only. As a logical extension, the model considers sloped terrain. By Assumption 3.3, the vehicle motion resistance force F_{MR} on sloped terrain can be expressed as

$$F_{MR} = \varepsilon N \tag{3.41}$$

where $N = mg\cos\phi$. Therefore, the towing force F_{TOW} required to keep the vehicle moving at constant speed on sloped terrain surfaces ($\phi > 0$) is expressed as

$$F_{TOW} = F_{MR} + mg\sin\phi = \varepsilon mg\cos\phi + mg\sin\phi \tag{3.42}$$

where $mg\sin\phi$ represents the tangential component of the gravitational force. Thus, on sloped terrain the

towing force required to overcome friction is reduced due to a decrease in the normal force. It is evident that the total specific resistance ϵ is a function of the vehicle characteristics and soil type and is independent of the terrain slope and velocity. This observation follows from Assumption 3.3.

c. Vehicle Braking

With respect to the towed vehicle model, a significant factor in the analysis of resistive forces is the vehicle braking system. Vehicle braking tends to increase internal moving element friction (bearing friction) and results in an increase in energy losses during vehicle motion. Braking action is achieved by either applying the brakes directly or initiating engine braking; i.e., *downshifting*. It is important to describe, mathematically, the relative degree of braking in order to quantify the amount of energy losses.

Definition 3.10: For a vehicular system, the percentage of braking is defined by the *braking coefficient* λ and reflects a full range of conditions. At the two extremes, the braking system can be either fully engaged or disengaged. A braking system engagement that lies between the two limits is defined as *partial braking*. The range of braking coefficient values can be expressed quantitatively as

- Non-braking Condition: $\{ \lambda = 0 \}$,
- Partial Braking Condition: $\{ 0 < \lambda < 1 \}$,
- Full Braking Condition: $\{ \lambda = 1 \}$.

The braking coefficient λ is used in the towed vehicle model to adjust the resistive force due to internal moving element friction MEF. In a situation where the braking system is fully disengaged, the moving element friction force F_{MEF} is assumed to come entirely from the bearing friction generated by all moving components within the vehicle except the braking system. As the brakes are engaged, there is a corresponding increase in the overall internal moving element friction due to the contribution of the bearing friction resulting from contact of the braking surfaces. Although the vehicle brakes can be applied at any time during travel, it is assumed that the braking system will be engaged only in situations where vehicle motion is directed downhill. Therefore, the primary purpose of braking in the towed vehicle model is to maintain a constant speed on downhill slopes; i.e. to avoid *acceleration*. Assigning an appropriate braking coefficient λ for a particular path segment has the effect of *modelling* the slope. The greater the percentage of vehicle braking, the steeper the descent and the greater the total increase in the component of total specific resistance attributable to internal moving element friction MEF.

Having introduced the concept of vehicle braking and the corresponding braking coefficient, an extended definition of total specific resistance is developed that incorporates the increased friction effects resulting from braking system engagements.

Definition 3.11: For a vehicle moving at constant speed on a fixed slope ϕ, the total motion resistance is a function of the braking coefficient λ and is defined by the *motion resistance coefficient* κ expressed as

$$\kappa = \frac{F_{TOW} - mg\sin\phi}{N}. \tag{3.43}$$

The motion resistance coefficient κ represents the total specific resistance ε adjusted for any vehicle braking that may be required to maintain constant speed on downhill slopes.

With the above definitions, it is important that the *limits* of vehicle motion resistance κ be defined. The lower bound on vehicle motion resistance occurs on terrain with a slope that is gentle enough that braking is not required.

Definition 3.12: Given a vehicle being towed on an arbitrary terrain surface with braking system disengaged ($\lambda = 0$) and in high gear, the downhill slope angle at which the tension in the towing cable begins to decrease, that is, begins to become slack, is defined as the *critical coasting angle* ϕ_{CC}.

Traveling downhill on terrain slopes that exceed the critical coasting angle ($\phi > \phi_{CC}$) implies that the force moving the vehicle is due entirely to gravity. At this point, the towing force F_{TOW} is equal to zero. Conversely, traversing slopes that do not exceed the critical coasting angle ($\phi \le \phi_{CC}$) assumes the towing force is positive. Using Eq. (3.3a) and the definition of motion resistance force, the equation of motion at the instant when rolling begins is

$$\varepsilon mg\cos\phi_{CC} + mg\sin\phi_{CC} = 0, \tag{3.44}$$

or

$$\varepsilon mg\cos\phi_{CC} = -mg\sin\phi_{CC}. \tag{3.45}$$

Rearranging terms, Eq. (3.45) becomes

$$-\tan\phi_{CC} = \varepsilon \tag{3.46}$$

and therefore, the critical coasting angle can be expressed as

$$\phi_{CC} = -\tan^{-1}\varepsilon. \tag{3.47}$$

Given the numerical value for the critical coasting angle ϕ_{CC} derived empirically, the solution to Eq. (3.46) produces a lower bound on the vehicle motion resistance.

The upper bound on vehicle motion resistance occurs under conditions of full braking. In the limit, just as the brakes become locked, the vehicle motion is no longer due to rolling, but instead, to sliding. The component of total specific resistance attributable to internal moving element friction e_{MEF} is at its maximum value under conditions of unity braking.

Definition 3.13: The minimum towing force F_{TOW} required to initiate motion on an arbitrarily-sloped terrain surface with the braking system fully engaged ($\lambda = 1$) is proportional to the normal force N and represents the vehicle *static sliding resistance* μ_s, defined quantitatively as

$$\mu_s = \frac{F_{TOW} - mg\sin\phi}{N}. \tag{3.48}$$

Eq. (3.48) is analogous to the standard definition of the coefficient of static friction in [Ref. 40] which is concerned with impending motion. The force required to sustain motion is slightly less than the force required to initiate the motion which necessitates the extension of Definition 3.13.

Definition 3.14: The towing force F_{TOW} required to keep the vehicle moving at constant speed on an arbitrarily-sloped terrain surface with the braking system fully engaged ($\lambda = 1$) is proportional to the normal force N and represents the vehicle *dynamic sliding resistance* μ_d, expressed quantitatively as

$$\mu_d = \frac{F_{TOW} - mg\sin\phi}{N}. \tag{3.49}$$

Eq. (3.49) is analogous to the traditional definition of the coefficient of kinetic friction [Ref. 40]. The expressions for both forms of sliding resistance are simplified for the towed vehicle model by the following assumption.

Assumption 3.4: For vehicular systems, the static sliding resistance μ_s is equal to the dynamic sliding resistance μ_d and is designated as the vehicle sliding resistance μ. The vehicle sliding resistance μ is invariant with the slope ϕ of the terrain surface.

Based on Assumption 3.4, it is possible to develop an upper bound for the vehicle motion resistance. To accomplish this, it is first necessary to define a critical angle associated with the slope of the terrain surface.

Definition 3.15: Given a vehicle at rest on an arbitrarily-sloped terrain surface with the braking system fully engaged ($\lambda = 1$), the slope angle at which the vehicle begins to slide down the incline is defined as the *critical braking angle* ϕ_{CB}, expressed as

$$\phi_{CB} = -\tan^{-1}\mu. \tag{3.50}$$

The critical braking angle ϕ_{CB} in the towed vehicle model is analogous to the angle of kinetic friction described in [Ref. 40]. Thus, given an experimentally derived value ϕ_{CB} for a particular vehicle, if F_{TOW} is set to zero (slack towing cable), then an upper limit for vehicle motion resistance is given by

$$\mu = -\tan\phi_{CB} \tag{3.51}$$

where μ is equal to the vehicle sliding resistance.

Given the definitions for total specific resistance ε and vehicle sliding resistance μ, a unifying relationship is established based on the braking coefficient. This relationship determines the portion of the total vehicle motion resistance attributable to the various components of resistance, as a result of braking system engagement. From the above discussion, it is observed that the motion resistance coefficient κ has a lower bound equal to the total specific resistance ε and an upper bound equal to the vehicle sliding resistance μ written as

$$\varepsilon \le \kappa \le \mu \tag{3.52}$$

where κ is computed as a weighted sum of the two resistances, defined as

$$\kappa = \varepsilon + \lambda(\mu - \varepsilon), \quad 0 \le \lambda \le 1. \tag{3.53}$$

The relationship of the vehicle motion resistance coefficient κ to the braking coefficient λ can be illustrated by examining the three relevant cases.

Case 1: Braking System Disengaged

When there is no braking, $\lambda = 0$, and the vehicle motion resistance coefficient is written as

$$\kappa = \kappa_{MIN} = \varepsilon. \tag{3.54}$$

The total resistance due to internal moving element friction is minimized in the absence of any additional friction caused by the vehicle braking system.

Case 2: Braking System Fully Engaged

For full braking conditions, $\lambda = 1$, and the vehicle motion resistance coefficient is defined as

$$\kappa = \kappa_{MAX} = \mu. \tag{3.55}$$

The total resistance due to internal moving element is maximized in the presence of additional bearing friction caused by the vehicle braking system.

Case 3: Braking System Partially Engaged

For partial braking conditions, $0 < \lambda < 1$, the vehicle motion resistance coefficient is expressed as

$$\varepsilon < \kappa < \mu. \tag{3.56}$$

The actual value of the motion resistance coefficient κ is computed using the appropriate percentages obtained from Eq. (3.53) and the upper and lower bound definitions from Eqs. (3.54) and (3.55).

Having described the conditions of vehicle braking, a relationship can be established between the slope angle ϕ, the critical coasting angle ϕ_{CC}, and the vehicle motion resistance coefficient κ. Given that a path segment S is derived from a position vector $\vec{s}$ which forms an angle ϕ with the reference axis or datum, it is possible to develop a *symbolic interpretation* of the path segment.

- Uphill: $\{ 0 < \phi < 90 \}$
- Level: $\{ \phi = 0 \}$
- Downhill: $\{ -90 < \phi < 0 \}$

With this symbolic interpretation, the following categories are defined.

- Uphill: $(\kappa = \kappa_{MIN} = \varepsilon)$ No critical coasting angle
- Level: $(\kappa = \kappa_{MIN} = \varepsilon)$ No critical coasting angle
- Downhill-1: $(\kappa = \kappa_{MIN} = \varepsilon)$ $\{ \phi_{CC} < \phi < 0 \}$
- Downhill-2: $(\kappa > \varepsilon)$ $\{ -90 < \phi < \phi_{CC} \}$

The partitioning of downhill path segments into two groups is important for energy-based path planning, and is formalized in the following definition.

Definition 3.16: A vehicle traversing a downhill path segment S with slope angle ϕ greater than the critical coasting angle ϕ_{CC} defines a *braking episode*. Conversely, a vehicle traversing a downhill path segment S with slope angle ϕ less than the critical coasting angle ϕ_{CC} defines a *non-braking episode*.

In addition to the symbolic interpretation of terrain, the vehicle motion resistance coefficient κ plays an integral role in the calculation of energy costs as described in the following section.

4. Energy Cost

For path-planning applications using the towed vehicle model, the cost function is based on the parameter of energy. From the basic definition of energy given in Section III.B.2, the energy cost represents the towing force F_{TOW} applied to an arbitrary vehicle over a specified distance d. The concept of vehicle energy cost is developed in two phases; a *local* approach involving a single path segment, and a *global* approach involving multiple, connected path segments.

a. Local Energy Cost

The energy required to overcome the effects of vehicle motion resistance and the force of gravity across a single path segment S represents a local energy cost. Substituting the vehicle motion resistance force F_{MR} into Eq. (3.17), results in a more specific definition of energy cost, expressed as

$$U_{s_1 \to s_2} = F_{MR}\, d + mg \sin\phi d. \tag{3.57}$$

Eq. (3.57) separates the energy cost attributable to the vehicle motion resistance from the energy cost associated with gravity. The energy cost due to motion resistance can be expressed in terms of the projected path distance D which gives rise to the following definition.

Definition 3.17: The energy cost associated with the application of the tangential component of the towing force across a projected path segment PS of distance D represents the *resistance energy cost* R defined as

$$R = \kappa mgD. \tag{3.58}$$

It is noted that the expression of the resistance energy cost R in the form defined by Eq. (3.58), is possible due to the cancellation of cosine terms associated with the normal force and calculation of the projected path distance. The resistance energy cost R is associated with the horizontal component of the total impedance to motion for vehicular travel with the vertical component resulting from the gravitational force and potential-energy considerations. As this point, a distinction can be made between the tangential motion axis described previously and the horizontal axis on which the resistance energy cost is defined. The latter (horizontal axis) contains the path segment that is the projection of the path segment contained in the former (tangential motion axis). This distinction is fundamental to the towed vehicle model since it facilitates the computation of motion resistance along a single axis. Thus, the resistance energy cost R is a function of the vehicle motion resistance coefficient κ and the straight-line distance D projected along the horizontal axis. Figure 3.3 illustrates the two components of energy cost for a single path segment. The

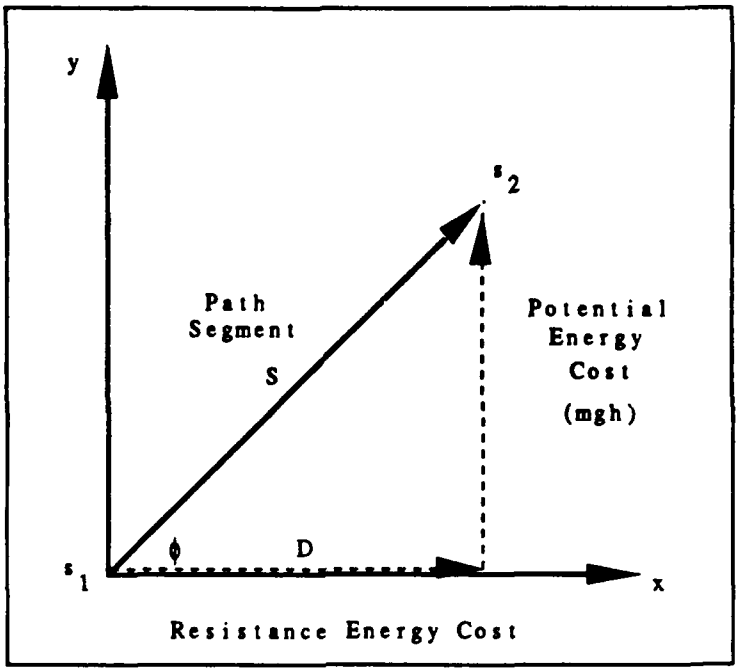

Figure 3.3 Components of Energy Cost

concept of resistance energy cost is formalized with a simplifying assumption and the following theorem.

Assumption 3.5: The vehicle motion resistance coefficient κ is invariant across a path segment S and its associated projected path segment.

Theorem 3.1: If $\vec{s}$ is the position vector defining the path segment S, then the energy cost of vehicle travel between the initial and final points defining the path segment is equal to the sum of the resistance-energy cost R for the projected path segment and the potential-energy change mgh between the two endpoints of the path segment, and is defined quantitatively as

$$U_{s_1 \to s_2} = R + mgh = mg(\kappa D + h) \tag{3.59}$$

where $R = mg\kappa D$ represents the energy losses due to Coulomb friction for the path segment and h is the change in elevation over the path segment.

Proof: From Eq. (3.3b), the normal force N can be rewritten as

$$N = mg\cos\phi \tag{3.60}$$

and therefore,

$$F_{MR} = \kappa mg\cos\phi. \tag{3.61}$$

Substituting into Eq. (3.23), the basic energy equation can be expressed in terms of resistive and braking coefficients as

$$U_{s_1 \to s_2} = \kappa mg\cos\phi\left[\frac{D}{\cos\phi}\right] + mgh \tag{3.62}$$

or equivalently,

$$U_{s_1 \to s_2} = \kappa mgD + mgh. \tag{3.63}$$

QED.

Eq. (3.63) represents the total energy required for a vehicle to overcome the effects of friction and gravity and maintain constant speed on an arbitrary path segment S with slope ϕ. From Theorem 3.1, the following generalization can be made with regard to the limits of the motion resistance coefficient κ and resistance energy cost R.

Corollary 3.1: For a given path segment S the resistive forces acting on the vehicle have a lower bound of $\kappa_{MIN}N$ and an upper bound of $\kappa_{MAX}N$ which results in an upper and lower bound for resistance energy cost R expressed as

$$R = \kappa_{MIN}mgD, \tag{3.64}$$

and

$$R = \kappa_{MAX}mgD. \tag{3.65}$$

The concept of local energy cost is extended by defining vehicle travel over multiple, connected path segments.

b. Global Energy Cost

For simple plane motion, Eq. (3.12) is extended by introducing the concept of multiple path segments, and the notion of connectivity between path segments. Figure 3.4 illustrates an instance of a multiple path segment.

Definition 3.18: Two path segments S_i and S_j with endpoints $\{s_{i1}, s_{i2}\}$ and $\{s_{j1}, s_{j2}\}$, respectively, are *connected*, denoted by $S_i \longleftrightarrow S_j$, if they share a common endpoint; i.e., $s_{i1} = s_{j1}$, $s_{i1} = s_{j2}$, $s_{i2} = s_{j1}$, or $s_{i2} = s_{j2}$. Thus, any path segment S_k sharing a common endpoint with another path segment S_l, such that the common endpoint is the termination point for at most two path segments, is defined as a ***connected path segment.***

With this definition, it is possible to address the issue of multiple path segment traversal and the total distance covered during vehicle motion.

Definition 3.19: A finite set of *connected path segments* $\{S_1, S_2, \ldots, S_n\}$, $n>1$, such that $S_1 \longleftrightarrow S_2$, $S_2 \longleftrightarrow S_3$,..., $S_{n-1} \longleftrightarrow S_n$, defines a *global path GP*. The length of the set of connected path segments is designated as the *global path distance* d_g and is the sum of the lengths of the individual path segments, expressed quantitatively as

$$d_g = \sum_{i=1}^{n} d_i. \tag{3.66}$$

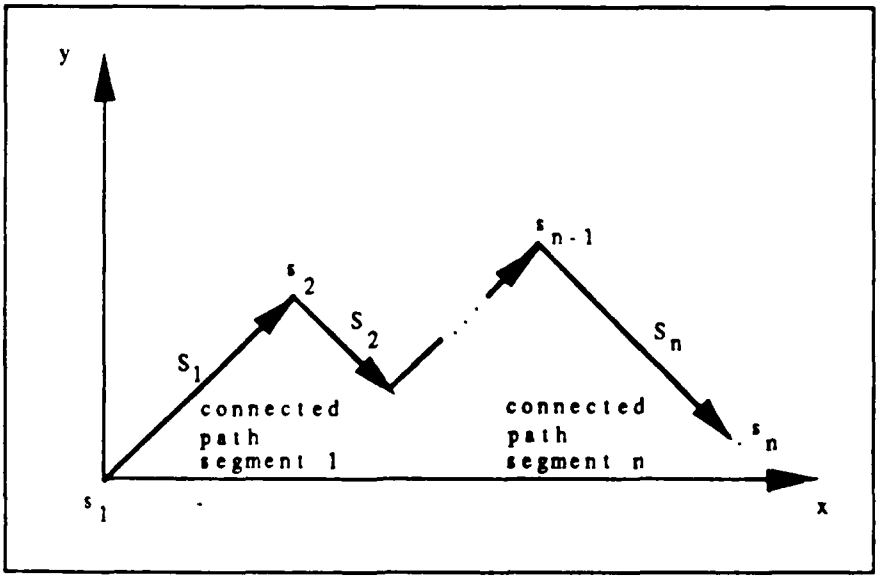

Figure 3.4 Multiple Path Segments

Thus, the total work done over a global path GP is defined as

$$U_{1\to n} = \sum_{i=1}^{n} F_{TOW_i} d_i = F_{TOW_1} d_1 + \cdots + F_{TOW_n} d_n \tag{3.67}$$

where each component term in the summation represents the work done over an individual *path segment* S_i. Substituting the equivalent opposing forces, Eq. (3.67) becomes

$$U_{1\to n} = \sum_{i=1}^{n} (F_{MR_i} + mg\sin\phi_i) d_i = (F_{MR_1} + mg\sin\phi_1) d_1 + \cdots + (F_{MR_n} + mg\sin\phi_n) d_n . \tag{3.68}$$

The resulting expression for the total work done by the towing force F_{TOW} over the global path of connected path segments is

$$U_{1\to n} = (F_{MR_1} d_1 + mg\sin\phi_1 d_1) + \cdots + (F_{MR_n} d_n + mg\sin\phi_n d_n). \tag{3.69}$$

Simplifying Eq. (3.69) results in the expression

$$U_{1\to n} = \sum_{i=1}^{n} F_{MR_i} d_i + mg \sum_{i=1}^{n} \sin\phi_i d_i \tag{3.70}$$

or equivalently,

$$U_{1\to n} = \sum_{i=1}^{n} F_{MR_i} d_i + mg \sum_{i=1}^{n} \Delta h_i . \tag{3.71}$$

It is evident that the sum of the elevation changes for the path segments can be expressed globally as

$$\sum_{i=1}^{n} \Delta h_i = h_f - h_s = h_g \tag{3.72}$$

where h_s and h_f represent the respective terrain elevations at the start and finish of vehicle travel. Using this definition, Eq. (3.71) can be rewritten as

$$U_{1\to n} = \sum_{i=1}^{n} F_{MR_i} d_i + mgh_g \tag{3.73}$$

and, thus, defines a *global energy equation* for multiple path segments. With this definition, it is evident that the global energy equation has both conservative and nonconservative components that represent the total change in potential energy and the total mechanical losses due to friction forces acting over the entire path. The results of Theorem 3.1 are extended to compute the energy costs for vehicle travel over a finite set of connected path segments.

Theorem 3.2: If $\{S_1, S_2, \ldots, S_n\}$ is the set of connected path segments defining a global path GP, then the global energy cost of vehicle travel between the initial point on path segment S_1 and the final point on path segment S_n is equal to the sum of the resistance energy costs R_i for all projected path segments and the global potential-energy change mgh_g, and is expressed as

$$U_{1\to n} = \left(\sum_{i=1}^{n} R_i\right) + mgh_g = mg\left[\left(\sum_{i=1}^{n} \kappa_i D_i\right) + h_g\right] \tag{3.74}$$

where each $R_i = mg\,\kappa_i D_i$ represents the energy losses due to Coulomb friction for an individual projected path segment and mgh_g is the elevation difference between the initial and final position of the vehicle.

Proof: The proof is a trivial extension of the singular path case expressed by Theorem 3.1. Let the resistive force for path segment S_i be represented as $F_{MR_i} = \kappa_i mg\cos\phi_i$. Substituting this expression and the definition of projected path distance, Eq. (3.22), into Eq. (3.73), results in the expression

$$U_{1\to n} = \sum_{i=1}^{n} (\kappa_i mg\cos\phi_i)\left[\frac{D_i}{\cos\phi_i}\right] + mgh_g \tag{3.75}$$

or equivalently,

$$U_{1\to n} = \left(\sum_{i=1}^{n} R_i\right) + mgh_g = mg\left(\sum_{i=1}^{n} \kappa_i D_i\right) + mgh_g. \tag{3.76}$$

QED.

The preceding theorem is designated as the *energy cost separation theorem*. From the proof of Theorem 3.2, the following generalizations can be made with respect to the slope ϕ of the terrain and the global path traversed by the vehicle.

Corollary 3.2: Resistance-energy cost R is a function of vehicle motion resistance κ, vehicle weight mg, and the projected path distance D, and is, therefore, independent of terrain slope ϕ.

Corollary 3.3: The global potential-energy change mgh_g for a vehicle traveling between any two points on the terrain, described by a global path GP is a function of the initial and final position of the vehicle, and is therefore, a constant, independent of the vehicle path.

The results of Corollaries 3.2 and 3.3 are fundamental to the towed vehicle model and the solution to the global, minimum-energy, path-planning problem. Evidently, for plane motion, the problem of finding minimum-energy paths can be viewed simply as a function of the vehicle motion resistance coefficient κ and the total straight-line distance D of the connected path segments projected onto the horizontal axis.

The actual computation of the coefficient of motion resistance κ_i for a given path segment depends on the value of the braking coefficient λ. While the value of λ_i for a particular path segment can be measured empirically, an alternative method is proposed.

Theorem 3.3: If for a given path segment S_i the traversing slope ϕ_i exceeds the critical coasting angle ϕ_{CC}, that is, $\lambda > 0$ and the path is classified as a braking episode, then the resistive-energy cost R_i is equal to the potential-energy loss $-mg\,\Delta h$.

Proof: For braking episodes, the equation of motion is expressed as

$$F_{TOW} = \kappa mg\cos\phi + mg\sin\phi = 0. \tag{3.77}$$

Solving Eq. (3.77) for κ, the value for the coefficient of motion resistance is expressed as

$$\kappa = -\tan\phi. \tag{3.78}$$

Substituting Eq. (3.78) into Eq. (3.63) the energy equation can be rewritten as

$$U_{s_1 \to s_2} = -mgD\tan\phi + mgh, \tag{3.79}$$

where $-D\tan\phi = -\Delta h$. Since the towing force is zero in braking regions, the total energy equation can be expressed as

$$U_{1\to n} = \varepsilon mg\sum_{i=1}^{k} D_i + mg\sum_{i=1}^{k}\Delta h_i \tag{3.80}$$

where the k path segments are non-braking episodes. Eq. (3.80) can be rewritten as

$$U_{1\to n} = \varepsilon mg\sum_{i=1}^{k} D_i + mg\sum_{i=1}^{n}\Delta h_i - mg\sum_{i=1}^{j}\Delta h_i, \tag{3.81}$$

where the j path segments are braking episodes and the n path segments are the combined braking and non-braking episodes.

Rearranging terms, Eq. (3.81) becomes

$$U_{1\to n} = \varepsilon mg\sum_{i=1}^{k} D_i + mg\sum_{i=1}^{j}|\Delta h_i| + mg\sum_{i=1}^{n}\Delta h_i, \tag{3.82}$$

or equivalently,

$$U_{1\to n} = \varepsilon mg\sum_{i=1}^{k} D_i + mg\sum_{i=1}^{j}|\Delta h_i| + mgh_g. \tag{3.83}$$

QED.

Thus, Eq. (3.83) is a generalized equation that provides a methodology for computing resistive-energy costs for both braking and non-braking episodes while maintaining the global, path-invariant component for potential-energy cost. Specifically, the energy cost involves only *horizontal* distance traveled in non-braking episodes, only *vertical* distance traveled during braking episodes, and a path-independent constant. All three terms in Eq. (3.83) are slope independent.

C. MOTION IN THREE DIMENSIONS

Vehicle motion in two dimensions has been described and the corresponding energy cost equations developed. Now, the towed vehicle model is extended to the next level of complexity involving motion in three dimensions. In the three-dimensional representation, vehicle motion is constrained to the terrain surface and maintains the restriction of fixed, straight-line path segments. The extended model is described in two distinct phases: for restricted and unrestricted vehicle motion.

1. Restricted Three-dimensional Motion

The first phase of the extended towed vehicle model involves motion on a restricted, three-dimensional, *cylindrical surface*. A cylindrical surface is a special case of a *ruled surface* which is defined in [Ref. 45] as a surface that, for every point on the surface, there is at least one straight line passing through it that lies entirely in the surface. In general, a cylindrical surface is constructed by sweeping a straight line along a curve. To establish a frame of reference for cylindrical terrain, a three-dimensional Cartesian coordinate system is defined with the x and y axes representing the *reference* or *datum plane*, and the z-axis describing the terrain elevation according to the function $z = f(x, y)$. For a topographic map interpretation of the reference plane, the y-axis equates to compass direction north at an azimuth of zero degrees. From this baseline, subsequent azimuth readings are measured in a positive, clockwise direction. For the towed vehicle model, a restricted form of a cylindrical surface is defined from a geometric and topological perspective.

Definition 3.20: A restricted cylindrical surface generated by moving a straight line of length L parallel to itself along a global path GP represented by a set of connected path segments $\{S_1, S_2, \ldots, S_n\}$, $n \geq 1$, is defined as *cylindrical terrain*. Cylindrical terrain consists of a set of rectangular, connected planar surfaces $\{P_1, P_2, \ldots, P_n\}$, called *cylindrical terrain patches*. Each cylindrical terrain patch P_i has dimensions d_i by L, where d_i is the length of path segment S_i.

In order to define the boundary of a cylindrical terrain patch P the concept of an undirected line segment is introduced.

Definition 3.21: A straight line bounded by the two endpoints V_1 and V_2, where each endpoint is described by its Cartesian coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2), respectively, represents an undirected line segment and is defined as an *edge segment* E.

As with path segments, there is a notion of connectivity between edge segments.

Definition 3.22: Two edge segments E_i and E_j with endpoints $\{e_{i1}, e_{i2}\}$ and $\{e_{j1}, e_{j2}\}$, respectively, are *connected*, denoted by $E_i \longleftrightarrow E_j$, if the two edge segments share a common endpoint; i.e., $e_{i1} = e_{j1}$, $e_{i1} = e_{j2}$, $e_{i2} = e_{j1}$, or $e_{i2} = e_{j2}$. Thus, any edge segment E_k sharing a common endpoint with another edge segment E_l, such that the common endpoint is the termination point for at most two edge segments, is defined as a *connected edge segment*.

Thus, a cylindrical terrain patch P can be defined geometrically by the *plane equation* $Ax + By + Cz + D = 0$ and a set of four connected edge segments $\{E_1, E_2, E_3, E_4\}$ forming a bounded rectangular planar surface in three dimensions. Each edge segment E_i can participate in $1 \le n \le 2$ cylindrical terrain patches, depending on its location in the sequence of connected patches.

Figure 3.5 provides several views of a restricted cylindrical terrain surface generated from a global path GP.

To quantitatively describe the surface configuration of cylindrical terrain, it is necessary to extend the concept of slope and discuss spatial change in three dimensions. As previously stated, slope is a two-dimensional concept expressing the vertical change in land surface over a specified horizontal distance and is always measured between two points on the terrain. In three dimensions, the concept of slope can be generalized to that of the *gradient* representing the maximum rate of elevation change occurring on a surface measured at a particular point [Ref. 46]. Since the gradient represents spatial change in three dimensions, it can be viewed as a vector with two distinct components defined quantitatively in [Ref. 41] as

$$\nabla f = \left[\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y} \right] \tag{3.84}$$

where the partial derivatives of $z = f(x, y)$ represent the partial slopes in the x and y directions of the topographic map plane. The partial slopes in the x and y directions are termed the *x-slope* and *y-slope*, respectively. Thus, the maximum rate of change of the terrain surface at any (x, y) map location is defined as the *gradient magnitude* G_m and is expressed as

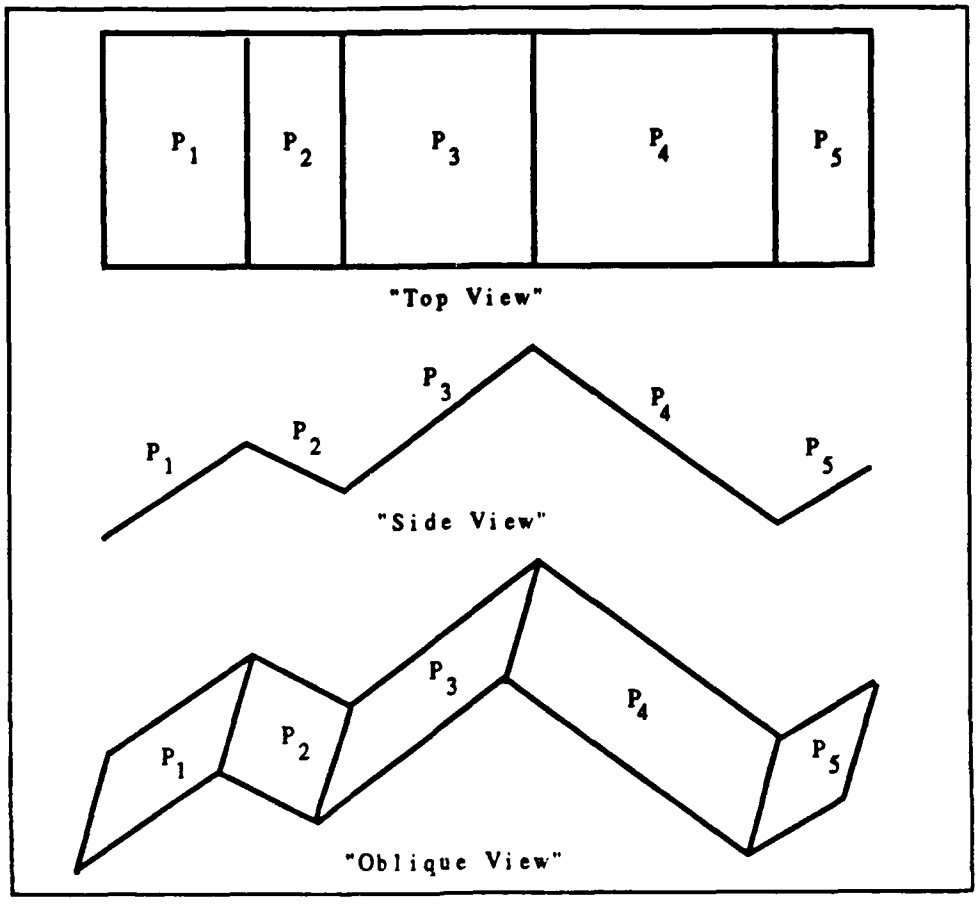

Figure 3.5 Restricted Cylindrical Terrain

$$G_m = \left[\left[\frac{\partial z}{\partial x} \right]^2 + \left[\frac{\partial z}{\partial y} \right]^2 \right]^{1/2}. \tag{3.85}$$

With this definition, the issue of maximum slope for cylindrical terrain patches can be addressed. The angle between a cylindrical terrain patch P and the topographic map plane is defined as the *gradient inclination angle* ϕ and is expressed as

$$\phi = \tan^{-1} \left[\left[\frac{\partial z}{\partial x} \right]^2 + \left[\frac{\partial z}{\partial y} \right]^2 \right]^{1/2}, \quad 0 \le \phi < 90. \tag{3.86}$$

The gradient inclination angle ϕ represents the maximum slope of a straight-line path over a rectangular surface patch with respect to a predefined reference plane.

For the towed vehicle model, it is important to know the direction of the spatial gradient as well as its magnitude. The direction of the spatial gradient is defined in [Ref. 46] as the *gradient azimuth angle* δ and is measured in the topographic map plane. The gradient azimuth angle δ is obtained by employing a *four quadrant* arctangent function defined as

$$\delta_1 = \tan^{-1} \frac{\partial z / \partial y}{\partial z / \partial x}, \quad \partial z / \partial x > 0, \tag{3.87a}$$

and

$$\delta_1 = 180 + \tan^{-1} \frac{\partial z / \partial y}{\partial z / \partial x}, \quad \partial z / \partial x < 0. \tag{3.87b}$$

Since the arctangent function uses the x-axis as its reference axis, it is necessary to adjust the gradient azimuth δ to correspond to the topographic reference frame defined for the towed vehicle model. The adjustment amounts to a simple 90 degree shift in the gradient azimuth δ for the appropriate quadrants and is expressed as

$$\delta = 90 - \delta_1, \quad \partial z / \partial x > 0, \tag{3.88a}$$

and

$$\delta = 90 + (360 - \delta_1), \quad \partial z / \partial x < 0. \tag{3.88b}$$

The resulting value of δ is measured in a clockwise direction from the y-axis (north) and represents the *direction of steepest descent.* Thus, for a given cylindrical terrain patch P, the gradient azimuth δ describes the orientation of the rectangular surface patch. For cylindrical terrain, in general, the orientation of patch P_j is equal either to the orientation of patch P_k, or to that orientation ± 180, for all $P_i \in \{P_1, P_2, \ldots, P_n\}$. It should be noted that the gradient inclination angle and gradient azimuth are strictly dependent on a particular cylindrical terrain patch, and are therefore, static in nature.

Given the definitions for maximum slope and orientation, it is appropriate to introduce the concept of path segments in three dimensions. To accomplish this task, it is necessary to extend Definition 3.1 which defines path segments for two-dimensional motion. If a point O defined by Cartesian coordinates (x_0, y_0, z_0) represents the origin of a three-dimensional coordinate system and the initial position of the vehicle, and point P defined by coordinates (x_p, y_p, z_p), represents the final position of the vehicle, the vector $\vec{s}$ describes a unique directed line segment $\vec{OP}$. If $\vec{s} = \vec{OP}$, then the coordinates of P are defined as components of $\vec{s}$ represented by the 3-tuple (s_1, s_2, s_3), and $\vec{s}$ is the position vector of point P.

The components of vector $\vec{s}$ can be expressed in terms of unit vectors defined for each of the three coordinate axes, i.e., $\vec{i} = (1,0,0)$, $\vec{j} = (0,1,0)$, and $\vec{k} = (0,0,1)$. Thus, $\vec{s} = (s_1, s_2, s_3)$ is given by $\vec{s} = s_1\vec{i} + s_2\vec{j} + s_3\vec{k}$ and is interpreted in the same manner as vector components in two dimensions.

Definition 3.23: For motion on restricted cylindrical terrain, a three-dimensional vector $\vec{s} = (s_1, s_2, s_3)$ lying entirely within a single cylindrical terrain patch, that is, satisfying the plane equation and the boundary conditions, is defined as a *path segment* S where s_1, s_2, and s_3 represent the components of $\vec{s}$ along the x, y, and z axes, respectively. The length or magnitude of $\vec{s}$, denoted by $|\vec{s}|$, is designated as the *terrain distance* d and is computed as the three-dimensional version of Euclidean distance defined as

$$d = (s_1^2 + s_2^2 + s_3^2)^{1/2}. \tag{3.89}$$

For the towed vehicle model, several important relationships exist between path segments, edge segments and the topographic map plane. The first relationship involves a path segment S and its projection onto the topographic map plane, and provides the distinction between overland distance and the distance measured on a two-dimensional topographic map.

Definition 3.24: Let $\vec{i} = (1,0,0)$ and $\vec{j} = (0,1,0)$ be unit vectors along the x and y axes and let $\vec{s} = (s_1, s_2, s_3)$ represent an arbitrary position vector defining a path segment S. The projection of $\vec{s}$ onto the topographic map plane, or the components of $\vec{s}$ in the $\vec{i}$ and $\vec{j}$ directions, denoted by $s_1\vec{i} + s_2\vec{j}$, is defined as a *projected path segment* PS. The length or magnitude of the projected path segment PS is designated as the *map distance* D and is expressed as

$$D^2 = |\vec{s}|\cos\phi = (\vec{s} \cdot \vec{i})^2 + (\vec{s} \cdot \vec{j})^2, \tag{3.90}$$

or equivalently,

$$D = \left[(x_2 - x_1)^2 + (y_2 - y_1)^2\right]^{1/2} = d\cos\phi \tag{3.91}$$

where ϕ is the angle between $\vec{s}$ and the topographic map plane.

The concept of path segment projection is readily extended to edge segments using Definition 3.21.

Definition 3.25: Let E be an arbitrary edge segment with endpoints V_1 and V_2 described by Cartesian coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2), respectively. The projection of edge segment E onto the topographic map plane is defined as a *projected edge segment* PE with endpoints V_1' and V_2' described by coordinates $(x_1, y_1, 0)$ and $(x_2, y_2, 0)$, respectively. The length of the projected edge segment PE is computed using Eq. (3.91).

For three-dimensional motion, the direction of travel is an important factor in the consideration of energy costs and is a key factor in the determination of vehicle stability. The direction of vehicle travel, as represented by a magnetic compass heading is defined as the *vehicle heading azimuth angle* ψ and is measured in a positive, clockwise direction from the previously established baseline, y-axis (north), in the topographic map plane. The vehicle heading azimuth angle ψ can also assume a full range of compass headings, i.e., $0 \leq \psi < 360$. But unlike the gradient inclination angle ϕ and gradient azimuth δ, azimuth angle ψ is strictly independent of any cylindrical terrain patch. There is an important relationship between the direction of vehicle travel ψ and a projected path segment PS. A projected path segment PS forms an azimuth angle with the y-axis in the topographic map plane that is equivalent to the vehicle azimuth ψ.

The relationship between the vehicle heading azimuth angle ψ and the gradient azimuth δ provides the foundation for *path segment classification.*

Definition 3.26a: Given a cylindrical terrain patch P with gradient and vehicle azimuths δ and ψ, respectively, if

$$\psi = \delta \tag{3.92a}$$

or

$$\psi = \delta + 180 \; mod \; 360 \tag{3.92b}$$

then the vehicle is traveling along a *gradient path.* The path segment S within the cylindrical terrain patch P defines a *gradient path segment.*

Satisfying the conditions of Eq. (3.92a) implies that the vehicle motion is in the downhill direction along the line of steepest descent. Fulfilling the conditions of Eq. (3.92b) implies that the vehicle motion is in the uphill direction along the line of steepest ascent.

Definition 3.26b: Given a cylindrical terrain patch P with gradient and vehicle azimuths δ and ψ, respectively, if

$$\psi = \delta + 90 \; mod \; 360 \tag{3.93a}$$

or

$$\psi = \delta + 270 \; mod \; 360 \tag{3.93b}$$

then the vehicle is traveling along a *contour path.* The path segment S within the cylindrical terrain patch P defines a *contour path segment.*

Traveling along a contour path is analogous to following a contour line on a topographic map. In a mathematical sense, it equates to traveling along a level curve or path perpendicular to the gradient at every point. A third category of path segment is defined for path segments that do not fall within the limits of gradient or contour paths.

Definition 3.26c: Given a cylindrical terrain patch P with gradient and vehicle azimuths δ and ψ, respectively, if

$$\psi = \delta + x \ mod\ 360 \quad \begin{cases} 0 < x < 90 \\ 90 < x < 180 \\ 180 < x < 270 \\ 270 < x < 360 \end{cases} \tag{3.94}$$

then the vehicle is traveling along an *oblique path*. The path segment S within the cylindrical terrain patch P defines an *oblique path segment*.

Figure 3.6 provides examples of the three general path classifications.

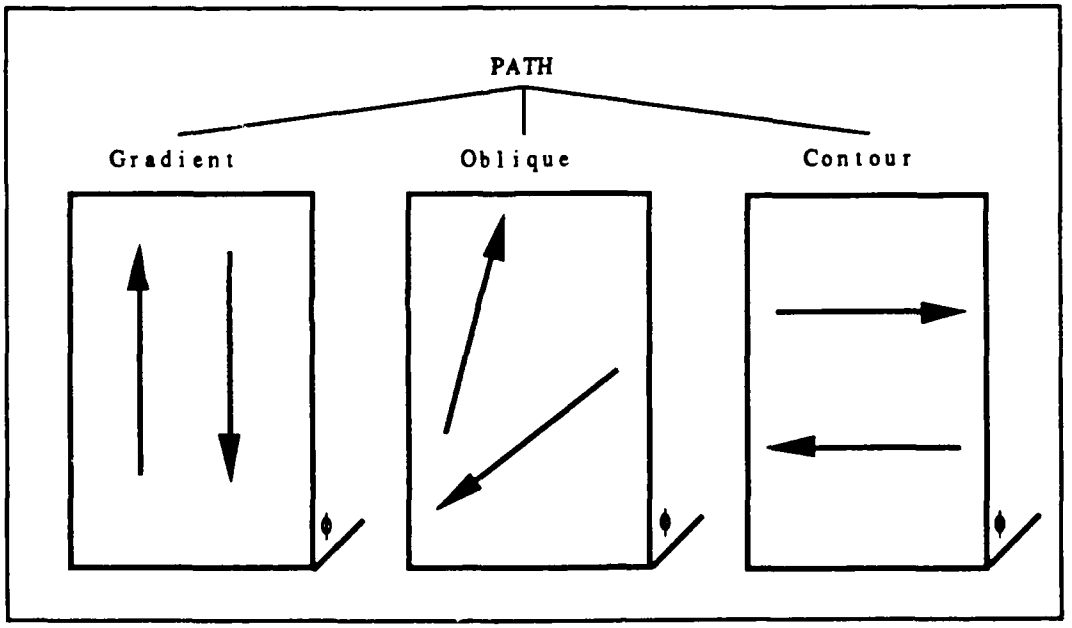

Figure 3.6 Path Segment Classification

Evidently, when a vehicle travels along a non-gradient path, it encounters a slope on a cylindrical terrain patch that is less than the actual gradient inclination angle ϕ of the patch. This inclination angle is important in the computation of energy costs for travel across natural terrain.

Proposition 3.1: The actual slope encountered by a vehicle in three-dimensional motion on cylindrical terrain is a function of the direction of travel and the surface configuration of the cylindrical terrain patch where the motion occurs and is defined as the *vehicle heading inclination angle* θ expressed quantitatively as

$$\theta = \tan^{-1} |\cos\psi \tan\phi|. \tag{3.95}$$

Proof: Let cylindrical terrain patch P defined by the set of connected edge segments $\{E_1, E_2, E_3, E_4\}$ contain a path segment S satisfying the plane equation and boundary conditions for the patch. Let a rectangular bounding box B consisting of the set of connected edge segments $\{E_5, E_6, E_7, E_8\}$ define a cylindrical terrain subpatch P_1 which encloses the two endpoints of the path segment S such that the two points define a pair of diagonal vertices of the bounding box B. Let L and d represent the width and length, respectively, of patch P and L_1 and d_1 represent the dimensions of subpatch P_1. Let d_2 represent the length of the path segment S and Z be defined as the maximum terrain elevation of subpatch P_1. Let the projection of segments S and E_5 onto the topographic map plane define projected segments PS of length D_2 and PE of length D_1, respectively.

There are three cases to be considered for gradient, contour, and oblique paths. Gradient and contour paths are special cases representing the limits of the more general (oblique path) case. Using the illustrations of Figure 3.7 the vehicle heading inclination angle θ is derived for the general case with respect to the first quadrant. The top view in Figure 3.7 represents the projection of the cylindrical terrain subpatch onto the topographic map plane. From this view, the vehicle azimuth ψ can be written as

$$\psi = \cos^{-1} \frac{D_1}{D_2}. \tag{3.96}$$

Viewing the subpatch from the side, the gradient inclination angle ϕ can be expressed as

$$\phi = \tan^{-1} \frac{Z}{D_1}. \tag{3.97}$$

The diagonal view is generated by conceptually slicing the subpatch along the path segment S. From this perspective, the vehicle heading inclination angle θ can be written as

$$\theta = \tan^{-1} \frac{|Z|}{D_2} \quad \theta \geq 0. \tag{3.98}$$

Solving Eqs. (3.96) and (3.97) for Z and D_2, respectively, and substituting the results into Eq. (3.98), it is evident that

$$\tan\theta = |\cos\psi \tan\phi| \tag{3.99}$$

or equivalently,

$$\theta = \tan^{-1} |\cos\psi \tan\phi|. \tag{3.100}$$

QED.

Eq. (3.100) can be generalized to account for the orientation of the cylindrical terrain subpatch P_1 and thereby extending the results to all four quadrants with a full range of compass headings. Incorporating the gradient azimuth δ the resultant equation for the four-quadrant vehicle heading inclination angle θ_1 becomes

$$\theta_1 = \tan^{-1} |(-\cos(\delta - \psi) \tan\phi)|. \tag{3.101}$$

Referring to Eq. (3.101), the term $-\cos(\delta - \psi)$ represents a *weighting factor* and can assume values from -1 to +1. Therefore, the absolute value is needed to make θ non-negative. The weighting factor determines the magnitude and sign of the four-quadrant heading inclination angle θ_1. Positive values of θ_1 indicate uphill travel while negative values of θ_1 imply downhill travel. For gradient paths, the four-quadrant heading inclination angle θ_1 assumes its maximum (absolute) value and is equal to $\pm$ the gradient inclination angle ϕ. For contour paths, where the vehicle is tilted along a side slope as it travels along a level curve, the heading inclination angle θ is equal to zero.

Proposition 3.2: Vehicle motion on restricted, three-dimensional, cylindrical terrain that follows a gradient path, that is, a connected set of gradient path segments, is equivalent to simple plane motion.

Proof: Let $\{S_1, S_2, \ldots, S_n\}$, $n \geq 1$, represent the set of connected path segments defining a global path GP such that each S_i is a gradient path segment obeying the criterion of Eqs. (3.92a) and (3.92b). Let $\{PS_1, PS_2, \ldots, PS_n\}$, represent the projected path segments generated from the projection of each S_i onto the topographic map plane. Let the plane, described by the equation $Ax + By + Cz + D = 0$, intersect the cylindrical terrain surface generated from the global path GP at the vehicle azimuth ψ and normal to the

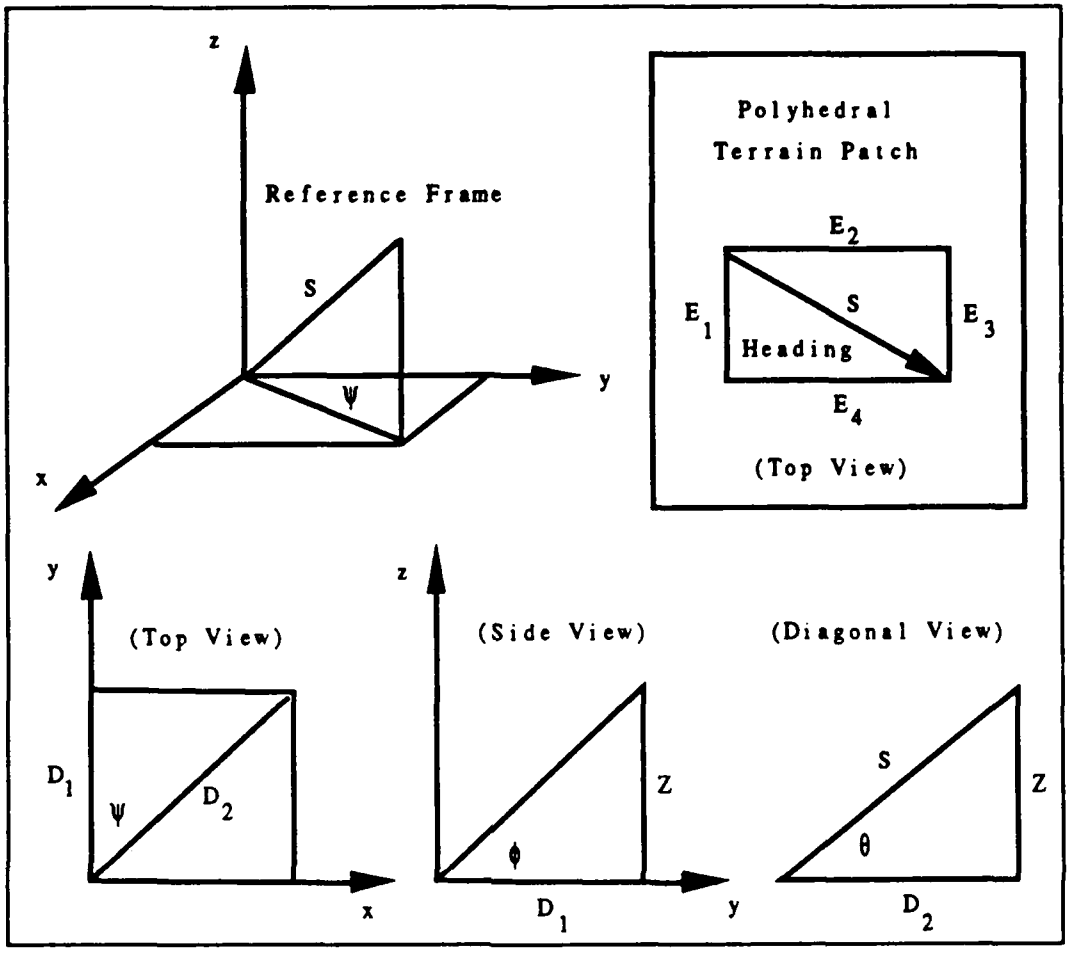

Figure 3.7 Heading Inclination Angle

topographic map plane. Each path segment S defined by its two endpoints s_1 and s_2 satisfies the plane equation and therefore, is fully contained within the designated plane. Since every path segment S within the set satisfies the criteria of the plane equation and the path segments are all connected, vehicle motion can be assumed to occur fundamentally in two dimensions which is equivalent to modelling the vehicle in simple plane motion.

QED.

Extending the range of motion on cylindrical terrain to include other than gradient paths, the following theorem can be stated:

Theorem 3.4: If vehicle motion on restricted, three-dimensional, cylindrical terrain includes a full range of possible headings, then the energy cost of vehicle travel can be expressed in terms of the heading inclination angle θ and the motion resistance force F_{MR} and is defined quantitatively as

$$U_{s_1 \to s_2} = F_{MR}\, d + mg \sin\theta d. \tag{3.102}$$

Proof: The proof is by substitution of Eq. (3.100) into the basic energy equation, Eq. (3.12).

Similarly for global paths, the energy equation can be expressed as

$$U_{1 \to n} = \sum_{i=1}^{n} F_{MR_i} d_i + mg \sum_{i=1}^{n} \sin\theta_i d_i \tag{3.103}$$

where $F_{MR_i} = \kappa_i mg \cos\phi_i$. Eq. (3.103) can be reduced using the relationships defined by Eqs. (3.71) and (3.72) to produce a global energy equation equivalent to Eq. (3.73) for multiple path segments on restricted cylindrical terrain.

It is observed that Eq. (3.103) is general in nature and applicable to the three classes of vehicle paths: gradient, contour, and oblique. The heading inclination angle θ can significantly alter the potential-energy component of the basic energy equation. For example, a vehicle traveling along an oblique path experiences a lesser change in elevation per unit distance than the same vehicle traveling along a gradient path. Using the heading inclination angle θ of the vehicle and the gradient inclination angle ϕ of the terrain surface, Eq. (3.102) can be rewritten with respect to each of the path types. Substituting the heading inclination angle θ into Eq. (3.22), an equivalent definition is provided for the path distance d in terms of the topographic map distance D expressed as

$$d = \frac{D}{\cos\theta}, \quad 0 \le \theta \le \phi. \tag{3.104}$$

For a vehicle traveling along a gradient path ($\theta = \phi$), the energy equation is equivalent to Eq. (3.57). The corresponding resistance energy cost R is defined by Eq. (3.58). A vehicle traveling along a contour path on restricted cylindrical terrain experiences no change in elevation during its motion; i.e., its heading inclination angle θ is equal to zero. Therefore, the potential energy is invariant along the path and can be ignored. The basic energy equation in this situation is a special case of the general form of Eq. (3.102) and becomes

$$U_{s_1 \to s_2} = F_{MR}\, d. \tag{3.105}$$

All motion resistance is attributed to the friction forces acting in the tangential direction to motion. Using

the same methodology employed for plane motion, the energy cost can be defined with respect to the two-dimensional, topographic map plane. Eq. (3.105) can be rewritten in terms of the map distance D as

$$U_{s_1 \to s_2} = \kappa mg \cos\phi \left[\frac{D}{\cos\theta} \right] \tag{3.106}$$

or simply

$$U_{s_1 \to s_2} = \kappa mg \cos\phi D. \tag{3.107}$$

For oblique paths, the energy equation is expressed as

$$U_{s_1 \to s_2} = \kappa mg \cos\phi \left[\frac{D}{\cos\theta} \right] + mgh = \kappa mgD \left[\frac{\cos\phi}{\cos\theta} \right] + mgh. \tag{3.108}$$

Physically, the term $\cos\phi/\cos\theta$ represents the ratio of the cosine of the terrain slope to the cosine of the slope experienced by the vehicle on a safe traversal. It is noted that the resistance energy cost R, defined as κmgD for gradient paths, is modified slightly for non-gradient paths. This modification is due to a "cosine effect" on the magnitude of the normal force N resulting from the gradient inclination angle ϕ and the orientation of the vehicle on the slope. The cosine effect is limited by the minimum possible heading inclination angle that the vehicle can traverse on a sideslope before overturning.‡ For example, on relatively steep slopes, cosine ϕ is smaller than on more gentle slopes but the path cannot deviate much from the gradient. Thus, the value of cosine ϕ is close to the value of cosine θ; that is, the angle between a gradient path and a possible oblique path is necessarily small. This situation implies that on steeper slopes the vehicle has a much smaller range of oblique headings permissible before to experiencing a catastrophic overturn failure. On slopes that are less steep, cosine ϕ is larger and the path can deviate more from the gradient. But the maximum value for cosine θ is still 1. Thus, the range of permissible oblique headings increases accordingly. Figure 3.8 illustrates the relationship between the vehicle azimuth angle and the slope of the terrain surface for non-gradient paths. Eq. (3.108) shows that the cosine terms effectively reduce the magnitude of the normal force. However, due to the limitations discussed above, the reduction is less than five percent on the average for a wide class of military vehicles. For the towed vehicle model, the cosine effect is assumed to affect the resistance energy cost R for vehicles traveling oblique or contour paths minimally and is, therefore, ignored in the overall computation

‡ Minimum slope occurs where vehicle heading differs maximally from gradient heading

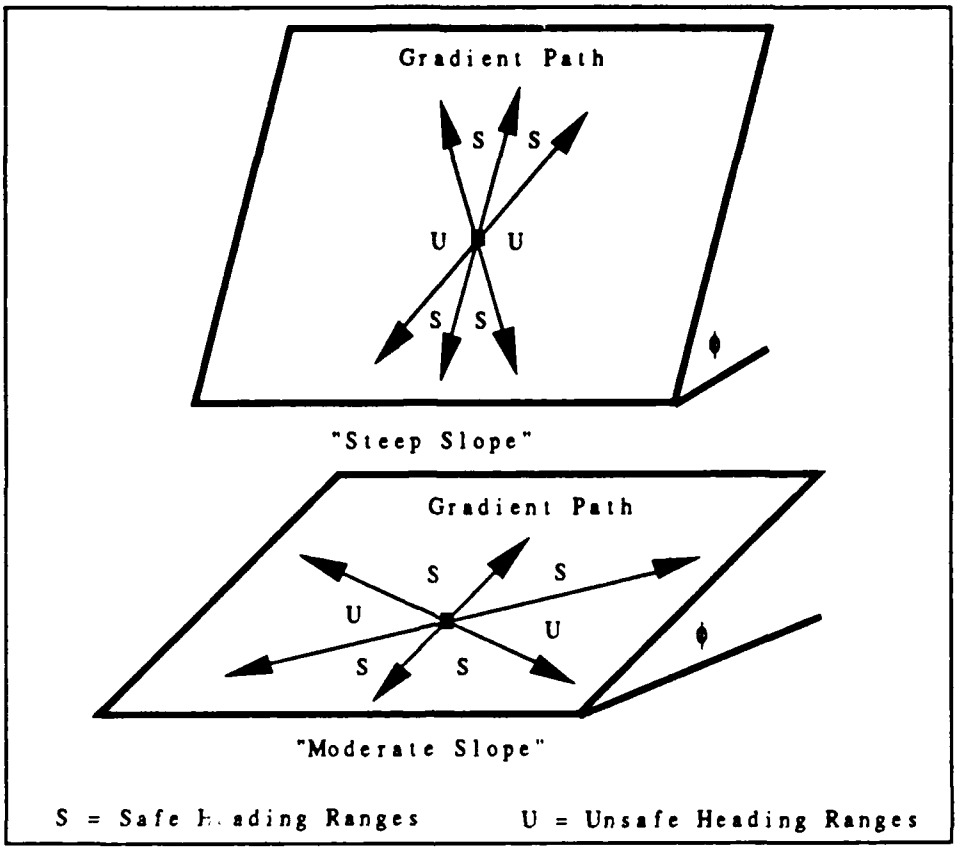

Figure 3.8 Cosine Effect on Non-gradient Paths

of minimum-energy paths. With this assumption, the energy equation for gradient paths is sufficient to handle all classes of paths. The above analysis justifies the application of the *energy cost separation theorem* to the three-dimensional case.

2. Unrestricted Three-dimensional Motion

To effectively model the complexity of natural landforms, a representative sampling of elevation data points is required. The data set, which can be of varying degrees of resolution and accuracy, is traditionally expressed in the form of a digital terrain database as described in [Ref. 13]. The digital database provides the basis for the development of a solid geometric model which can be used to create an unambiguous, informationally complete, mathematical representation of the terrain structure. For the towed vehicle model, the definition of restricted cylindrical terrain is extended by constructing a generalized, three-dimensional, planar surface within the same Cartesian coordinate system.

To accomplish minimum-energy path planning using the *ray tracing approach*, as opposed to the classic grid-based *wavefront approach*, it is necessary to develop a terrain representation that considers regions of uniform gradient. In general, the property of uniform gradient implies that, from any position within a designated region, the terrain slope and orientation are constant within a designated threshold. The continuous nature of the terrain can be approximated using several different geometric models, all employing the uniform gradient property. These models can be viewed as a hierarchy of *polygonal uniform gradient regions* each at an increasing level of complexity. In general, the terrain surface is represented by an irregular *polyhedron* defined in [Ref. 45] as the arrangement of convex polygons such that a maximum of two polygons meet at an edge. In the simplest case, joining every three data points results in a set of n triangles forming the planar faces of the polyhedron.

Definition 3.27: A generalized, three-dimensional surface generated by taking a set of triangular surfaces $\{P_1, P_2, \ldots, P_n\}$, $n > 1$, and joining the surfaces along the edges according to the topological constraints of a polyhedral structure, is defined as *triangular polyhedral terrain*. Each piecewise flat surface patch P_i is designated as a *triangular polyhedral terrain patch*. A triangular polyhedral terrain patch P can be defined geometrically by the plane equation $Ax + By + Cz + D = 0$ and a set of three connected edge segments $\{E_1, E_2, E_3\}$.

A special case of triangular polyhedral terrain occurs when two or more of the triangles lie in the same plane.

Definition 3.28: A generalized three-dimensional surface generated by combining triangular polyhedral terrain patches satisfying the same plane equation into a set of convex, planar polygonal surfaces, $\{P_1, P_2, \ldots, P_n\}$, $n > 1$, and joining the surfaces along the edges according to the topological constraints of a polyhedral structure, is defined as *generalized polyhedral terrain*. Each piecewise flat surface patch P_i is designated as a *generalized polyhedral terrain patch*. A generalized polyhedral terrain patch P can be defined geometrically by the plane equation $Ax + By + Cz + D = 0$ and a set of n connected edge segments $\{E_1, E_2, \ldots, E_n\}$, $n \geq 3$.

For the towed vehicle model, it is possible to generalize the concept of polyhedral terrain and partition the map by aggregating data points with homogeneous properties. Specifically, the aggregation process includes the data points that exhibit a constant gradient property within a designated threshold.

Definition 3.29: The aggregation of n constant gradient terrain data points, $n \geq 1$, within a designated threshold defines a *polygonal terrain patch* P. The set of n polygonal terrain patches $\{P_1, P_2, \ldots, P_n\}$, $n \geq 1$, is defined as *polygonal terrain*.

Figure 3.9 provides an example of a polygonal terrain surface constructed from individual polygonal terrain patches. The concept of projected edge segments can be applied to polygonal terrain surface patches.

Definition 3.30: The projection of a set of n connected edge segments $\{E_1, E_2, \ldots, E_n\}$, $n \geq 3$, representing the boundary of a polygonal terrain patch, onto the topographic map plane is defined as a *projected polygonal terrain patch* PL. The set of n projected polygonal terrain patches $\{PL_1, PL_2, \ldots, PL_n\}$, $n \geq 1$, is defined as *projected polygonal terrain*. The set of projected polygonal terrain patches $\{PL_1, PL_2, \ldots, PL_n\}$, $n \geq 1$, forms a *polygonal tiling* of the two-dimensional topographic map plane.

Each projected polygonal terrain patch possesses configuration characteristics (slope and orientation) that are essential to energy-based path planning. The resulting segmentation of the terrain is based solely on the

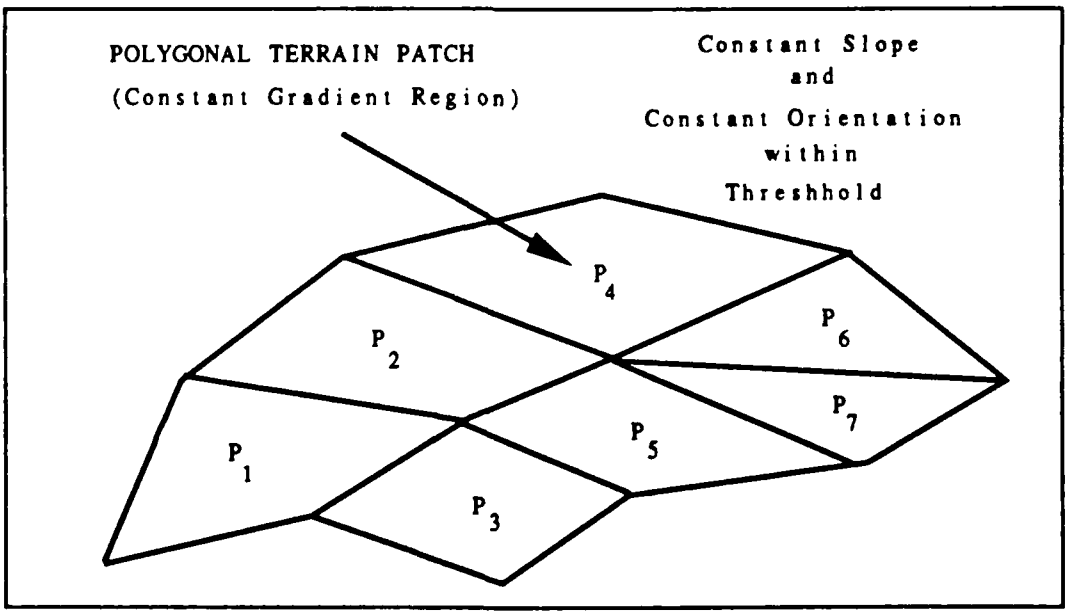

Figure 3.9 Unrestricted Three-dimensional Terrain

uniform gradient property and is not concerned with the topological constraints imposed on polyhedral terrain surfaces. The uniform gradient regions represent the fundamental set of objects participating in the search process for finding minimum-energy paths in natural terrain. In essence, the model views the path-planning problem from a two-dimensional perspective retaining significant three-dimensional information as required. Thus, the complexity of the problem is effectively reduced. A complete discussion of polygonal uniform gradient terrain patches is presented in Section III.E.

The energy equations developed for restricted cylindrical terrain surfaces are general in nature and applicable to motion on polygonal uniform gradient terrain. The principal distinction between the two types of terrain is the restriction placed on the shape and orientation of constituent surface patches. Polygonal uniform gradient terrain facilitates unrestricted three-dimensional motion on a surface which more closely approximates the complexity of natural terrain.

D. VEHICLE FAILURE MODES

For a given set of projected polygonal terrain patches $\{PL_1, \ldots, PL_n\}$, representing a set of n uniform gradient regions, it is possible to restrict vehicle movement over selected patches, or portions of selected patches, because of the surface configuration, surface composition or direction of travel. For the towed vehicle model, the restrictions on vehicle travel are designated as *motion constraints*. Motion constraints are related to the maximum slope capability of the vehicle and the angles at which the vehicle will overturn on a given slope. Motion constraints involving maximum slope capability depend upon the configuration or geometry of the terrain surface as well as the composition and physical state of the surface. Motion constraints relating to stability, in contrast, depend only on surface configuration and vehicle azimuth. The restrictions on movement within regions can significantly reduce the global map area to be searched in the process of finding a solution to the minimum-energy path-planning problem. It is noted that failure modes such as *hang-up failure* (HUF) and *nose-in failure* (NIF) as described by Bekker in [Ref. 25], are considered only in local path-planning problems.

1. Motion Constraints for Maximum Slope

Motion constraints for maximum slope exist in regions where the inclination angle of the terrain patch and the soil composition together cannot provide sufficient support for a particular vehicle. Therefore, sliding ensues. This type of motion constraint can be quantified using the concept of the critical braking angle ϕ_{CB}. As previously noted, the critical braking angle places an upper limit on the vehicle motion resistance coefficient κ. Exceeding the upper limit on the motion resistance coefficient κ_{MAX}

represents a situation in which the vehicle, with its brakes locked, slides down the sloped surface. This friction-force limitation, resulting from the soil-slope combination, is measured empirically and serves to eliminate entire terrain patches from potential traversal. The motion constraints for maximum slope provide the principal evaluation parameter in the identification and selection of regions of impermissibility or obstacle regions for purposes of terrain classification. Obstacle and non-obstacle regions are discussed in detail in Section III.E.

2. Motion Constraints for Stability

Motion constraints for stability are primarily concerned with the possibility of the vehicle overturning on a sloped surface. Previous formalizations of the criteria for determining vehicle stability are given in [Ref. 47] and [Ref. 48]. Following the same methodology as in the development of the energy cost equations, the problem of stability is first examined in two dimensions for vehicles in plane motion and subsequently extended to address stability-related motion constraints for unrestricted three-dimensional motion.

For the two-dimensional representation, the vehicle can be viewed as a rectangle with dimensions h x b, as Figure 3.10 illustrates. The distance b can either represent the length of the vehicle, if

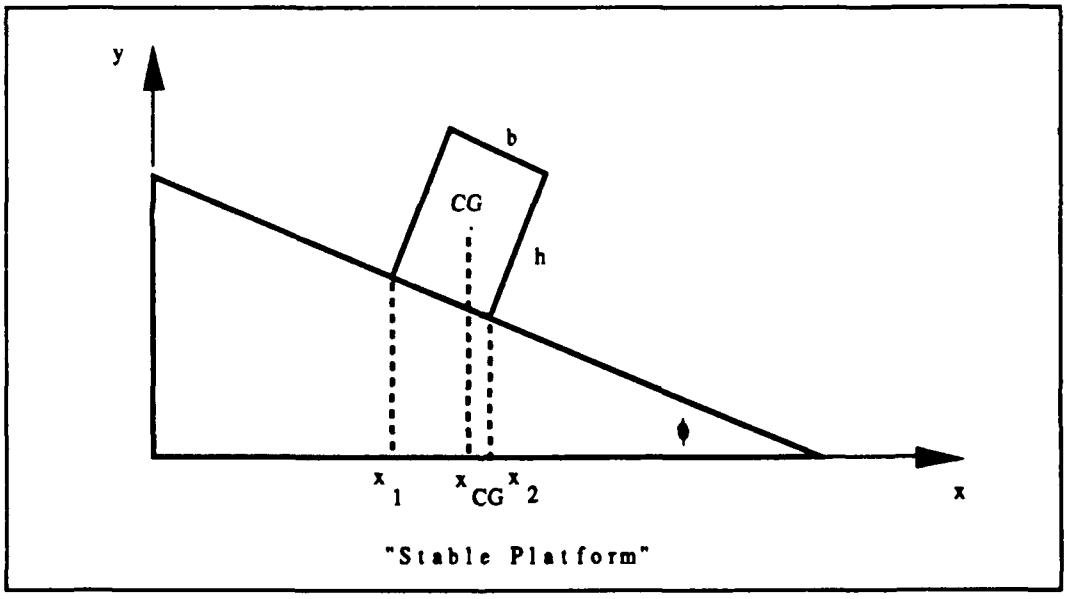

Figure 3.10 Two-dimensional Stability Model

traveling along a gradient path, or the width of the vehicle, if traveling along a contour path. Let the base of the rectangle be represented by the edge segment E with endpoints P_1 and P_2 described by Cartesian coordinates (x_1, y_1) and (x_2, y_2), respectively. Edge segment E is assumed to be collinear with the line defining the slope. Let the center of gravity of the vehicle CG with coordinates (x_{CG}, y_{CG}), be defined by the intersection of the two perpendicular bisectors of h and b. Given the above configuration, the projection of edge segment E onto the horizontal axis represents a projected edge segment PE with endpoints P_{01} and P_{02} described by coordinates $(x_1, 0)$ and $(x_2, 0)$, respectively. Similarly, the vertical projection of the vehicle center of gravity CG onto the horizontal axis is defined by the projection point P_0 with coordinates $(x_{CG}, 0)$. The shortest distance from the vertical projection of the center of gravity P_0 to the endpoints P_{01} and P_{02} of the projected edge segment PE represents the measure of *stability* for the vehicle and is expressed quantitatively as

$$MIN\left[SM_1, SM_2\right] \tag{3.109}$$

where $SM_1 = |x_1 - x_{CG}|$ and $SM_2 = |x_2 - x_{CG}|$. There is also the restriction that $x_1 < x_{CG} < x_2$. Evidently, the vehicle is at a point of maximum stability when the two distances are equal and becomes progressively less stable as one of the two distances decreases. The threshold between stability and instability is reached when the distance between P_0 and either endpoint is zero. A state of *instability* exists when $x_{CG} < x_1$ or $x_{CG} > x_2$. This situation occurs when the vertical projection of the center of gravity moves outside of the projected edge segment as illustrated in Figure 3.10. After stability in two dimensions has been described, the model is easily extended to three dimensions.

Assumption 3.6: An arbitrary vehicle with massless extensions of length L, width W, and height H has a center of gravity located at (x_{CG}, y_{CG}, z_{CG}).

By modifying the definitions of static stability for legged vehicles given in [Ref. 47] and [Ref. 48], corresponding definitions are developed for the towed vehicle model.

Definition 3.31: The base of a vehicle with massless extensions of length L and width W is represented by a set of four connected edge segments $\{E_1, E_2, E_3, E_4\}$ positioned in the same plane that defines the polygonal terrain patch. The rectangular polygon defined by these edge segments is designated as the *vehicle support boundary*.

Using Definition 3.25, the edges of the support boundary are projected onto the horizontal plane; i.e., the topographic map plane.

Definition 3.32: The projection of the set of edges $\{E_1, E_2, E_3, E_4\}$ representing the vehicle support boundary, onto the topographic map plane forms a set of connected edge segments $\{PE_1, PE_2, PE_3, PE_4\}$ defined as the *vehicle support pattern*.

Given a particular support boundary and support pattern defined in the polygonal and topographic map planes respectively, the vehicle center of gravity is vertically projected onto the horizontal plane. Again, by geometry, the distance from the vertical projection of the center of gravity to any point on the vehicle support pattern can be computed. Applying the definitions developed in [Ref. 47] and [Ref. 48] to the criteria for vehicle stability in three dimensions gives the following definition.

Definition 3.33: For an arbitrary vehicle represented in three dimensions by a set of massless extensions on a sloped terrain surface, the shortest distance from the vertical projection of the center of gravity to any point on the vehicle support pattern is defined as the *vehicle stability margin*.

In the three-dimensional model, it is evident that maximum stability results when the vertical projection of the center of gravity is at the exact center of the vehicle support pattern, that is, the point at which the stability margin reaches its maximum value. The stability margin is assumed to be positive when the vertical projection of the center of gravity is strictly within the support pattern and negative otherwise [Ref. 48]. Figure 3.11 illustrates the concept of vehicle stability in three dimensions.

Given the two types of motion constraints for vehicle travel over arbitrary terrain surfaces, a taxonomy of *vehicle failure modes* is developed. In general, a failure mode occurs when the vehicle attempts to travel at a heading that exceeds the associated constraints for maximum slope and stability. As a result, the vehicle is impeded from further movement across the particular terrain surface. The path and path segment classifications previously defined, provide a framework for analyzing the situations in which a vehicle failure can occur. For the towed vehicle model, three failure modes are defined.

Definition 3.34: The lack of sufficient friction force to keep a vehicle in a fixed position on a sloped terrain surface when traveling in the direction of maximum ascent or descent, represented by the heading azimuth $\psi = \delta$ or $\psi = \delta + 180 \; mod \; 360$, results in a *gradient failure* and is defined as a *gradient failure mode*.

As previously noted, with respect to motion constraints for maximum slope, a gradient failure represents a situation in which the vehicle slides down a sloped surface due to the magnitude of the incline and the composition of the soil. Gradient failures can also be attributed to stability considerations described in Bekker [Ref. 25] as longitudinal failures. However, for the towed vehicle model, it is assumed that a friction-force failure will occur prior to longitudinal overturn for any soil-slope combination when the vehicle is traveling along a gradient path.

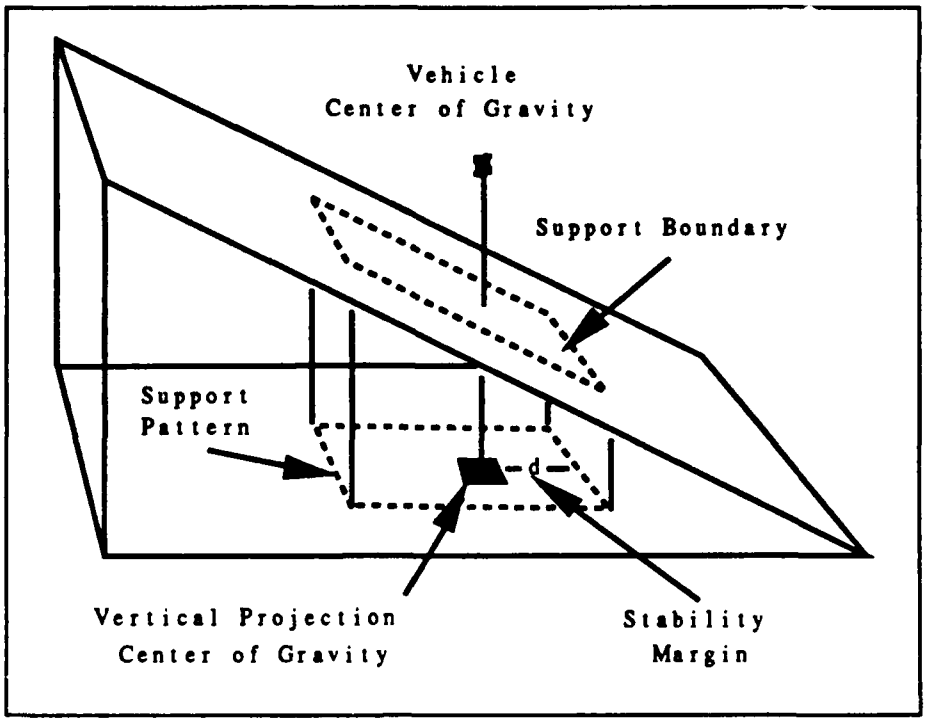

Figure 3.11 Three-dimensional Stability Model

The second type of failure occurs when the vehicle orientation is ± 90 degrees to the gradient azimuth δ of the terrain surface.

Definition 3.35: The lack of a sufficient margin of stability to prevent a vehicle from overturning on a sloped terrain surface when traveling along a level curve, represented by the heading azimuth $\psi = \delta + 90 \; mod \; 360$ or $\psi = \delta + 270 \; mod \; 360$, results in a *contour failure* and is defined as a *contour failure mode*.

Contour failures are only associated with stability-related motion constraints. Although it is theoretically possible to slide down a terrain surface while traveling along a level curve, it is assumed that the maximum angle at which overturn occurs is always less than the maximum angle at which a gradient failure occurs. If this is not the case, then the entire terrain patch is eliminated from consideration due to motion constraints for maximum slope, i.e., friction-force failure.

Definition 3.36: The maximum gradient inclination angle ϕ of a terrain surface at which a vehicle can safely travel along a contour path of that same surface is defined as the *critical stability angle* ϕ_{CS}.

For military vehicles typically involved in off-road travel, the critical stability angle ϕ_{CS} is described as the maximum side-slope angle and is established by military doctrine. The critical stability angle ϕ_{CS} is significantly smaller than the gradient inclination angle ϕ at which the actual overturn occurs. The magnitude of this difference is attributed to the size of the stability margin for a particular vehicle traveling at a contour heading on a given slope. In this situation, the stability margin is considered a safety factor that is employed to minimize the risk of a catastrophic overturn. With this definition, an observation can be made with regard to the maximum side-slope angle and the maximum gradient inclination angle for an arbitrary terrain patch. It is noted that the critical stability angle ϕ_{CS} is always less than the critical braking angle ϕ_{CB} for a vehicle traveling on an arbitrary terrain surface. This observation has significant implications for the towed vehicle model; that is, an entire class of terrain patches can be eliminated from the potential search area because of motion constraints for maximum slope (friction-force failure). The remaining terrain patches can be checked for possible stability problems.

The final failure mode is a combination of the previous two failure modes and can occur when the vehicle is traveling along an oblique path. Using the previously defined stability margin associated with the critical stability angle ϕ_{CS}, an equivalent safety factor is applied to a vehicle negotiating a sloped surface at an oblique heading angle.

Definition 3.37: The lack of a sufficient margin of stability to prevent a vehicle from overturning on a sloped terrain surface when traveling along an oblique path, represented by the heading azimuth $\psi = \delta + x \ mod \ 360, \ x \neq 0, 90, 180, 270$, results in an *oblique failure* and is defined as an *oblique failure mode.*

This type of failure implies that for an arbitrary terrain patch with no motion constraints due to friction force failure, there is a range of oblique headings that a vehicle can safely travel without overturning. Depending on the gradient inclination angle ϕ of the patch, the range of permissible headings can extend symmetrically from the gradient heading to the contour heading. As the slope gets progressively steeper, the range of permissible headings decreases in the direction of the gradient heading. Conversely, for a very gentle slopes, there is a much wider range of permissible headings, increasing in the direction of the contour heading. As with contour failures, oblique failures are only associated with insufficient stability.

Definitions 3.34, 3.35, and 3.37 provide a symbolic interpretation of the various classes of vehicle failure modes. For the towed vehicle model, it is important to relate vehicle failure modes to a vehicle azimuth, or heading, in order to avoid planning routes with a high probability of motion constraints.

Therefore, for each vehicle traversing a terrain patch, a determination must be made as to the ranges of headings that are *safe* and the ranges of headings that are *unsafe* within the patch. Since the motion constraints for maximum slope eliminate entire terrain patches due to friction-force considerations, the remaining focus is on stability failures. Using the previously defined compass designation for the topographic map plane with zero degrees due north, a critical heading angle is developed.

Definition 3.38: Let polygonal terrain patch P contain the support boundary of a selected vehicle and the corresponding support pattern in the topographic map plane. The vehicle azimuth angle, measured from the gradient azimuth angle δ (or $\delta + 180$ *mod* 360) in the x-y plane, represents the point at which the stability margin becomes less than the minimum allowable for safety considerations and is defined as the *critical stability azimuth* ψ_{CS}.

An initial critical stability azimuth for the vehicle being determined, the concept is extended to create a set of limiting bounds for a vehicle operating from a given position on a polygonal terrain patch.

Definition 3.39 The magnitude of the angle between the gradient azimuth δ (or $\delta + 180$ *mod* 360) and the critical stability azimuth ψ_{CS} measured in the topographic map plane is defined as the *stability offset* α where $0 < \alpha < 90$.

The stability offset represents the range of permissible headings for a particular vehicle on a specific terrain surface with respect to a gradient path on that same surface. For a polygonal terrain patch P with gradient inclination angle ϕ greater than the critical stability angle ϕ_{CS}, there are two sets of symmetric critical stability azimuths $\{\psi_{CS-1}, \psi_{CS-2}\}$, and $\{\psi_{CS-3}, \psi_{CS-4}\}$, representing the ranges of vehicle headings that are permissible for uphill and downhill travel without triggering a stability failure. For downhill travel ($\phi < 0$), the critical stability azimuths are expressed as

$$\psi_{CS-1} = \delta - \alpha \tag{3.110a}$$

and

$$\psi_{CS-2} = \delta + \alpha. \tag{3.110b}$$

Negative values of ψ_{CS-1} are converted to a compass orientation by adding 360 to the initial result. Values of ψ_{CS-2} that are greater than or equal to 360 are adjusted by subtracting a similar amount.

For uphill vehicle travel ($\phi > 0$), the critical stability azimuths become

$$\psi_{CS-3} = (\delta + 180 \; mod \; 360) - \alpha \tag{3.111a}$$

and

$$\psi_{CS-4} = (\delta + 180 \; mod \; 360) + \alpha. \tag{3.111b}$$

Corrections to critical stability azimuths for the proper compass orientation are computed as above. The stability offset provides a mechanism for constructing two sets of critical azimuths that define the limits of permissible heading ranges for a particular vehicle traveling across a uniform gradient region represented by a polygonal terrain patch. The critical stability azimuths within each set are symmetric about the downhill gradient azimuth δ or the uphill gradient azimuth $\delta + 180 \; mod \; 360$. Figure 3.12 provides an illustration of the critical stability azimuths and the impermissible ranges of headings for a polyhedral terrain patch.

To complete the discussion on heading constraints, one final set of critical azimuth angles is proposed. While not affecting vehicle stability, constraints on vehicle headings due to braking phenomenon

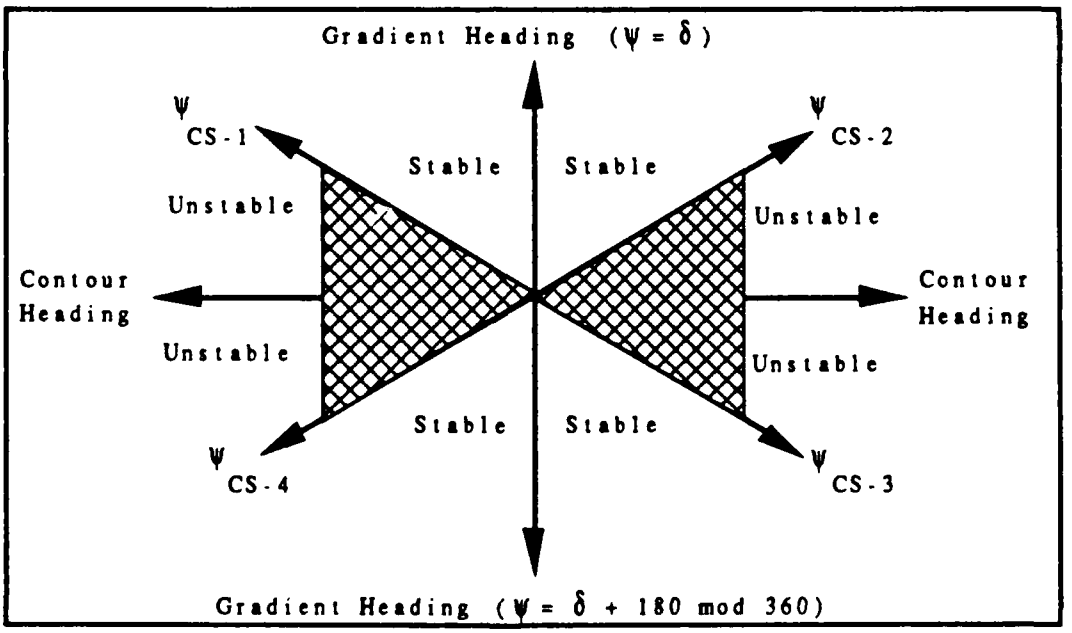

Figure 3.12 Critical Stability Headings

are considered significant to the minimum-energy path-planning problem. The primary concern for vehicles traveling an oblique path on a downhill slope is the azimuth angle at which braking is initiated in order to maintain constant speed, that is, to avoid acceleration.

Definition 3.40: Given a vehicle being towed on a polygonal terrain patch P the vehicle azimuth ψ measured from the gradient azimuth δ in the topographic map plane, representing the point at which the tension in the towing cable begins to decrease as the vehicle starts to roll downhill (a three-dimensional interpretation of Definition 3.12), is defined as the *critical braking azimuth* ψ_{CB}.

An initial critical braking azimuth from Definition 3.40 having been determined, the concept is extended to create a set of braking constraints for a designated vehicle on a given polygonal terrain patch.

Definition 3.41: The magnitude of the angle between the gradient azimuth δ and the critical braking azimuth ψ_{CB} measured in the topographic map plane, is defined as the *braking offset* β, where $0 < \beta < 90$.

The braking offset β represents the range of braking headings for a particular vehicle on a specific terrain surface with respect to the gradient azimuth on that same surface. For a polygonal terrain patch P with gradient inclination angle ϕ there is a set of symmetric critical braking azimuths $\{\psi_{CB-1}, \psi_{CB-2}\}$, representing the range of vehicle headings where braking can occur during downhill travel. The critical braking headings can be expressed quantitatively as

$$\psi_{CB-1} = \delta - \beta \tag{3.112a}$$

and

$$\psi_{CB-2} = \delta + \beta. \tag{3.112b}$$

All values are converted to the proper grid compass orientation using the approach outlined above. A vehicle heading within the range of headings bounded by the critical braking headings ψ_{CB-1} and ψ_{CB-2}, is designated as a *braking heading* ψ_{BR}. A vehicle heading that is not within the range of braking headings is designated as a *non-braking heading* ψ_{NB}. Figure 3.13 provides an illustration of the critical braking azimuths and the range of braking headings for a polyhedral terrain patch.

From Figures 3.12 and 3.13, it is evident that there can be up to six critical headings defined relative to a specific vehicle position on a polyhedral terrain patch, i.e., four for stability and two for braking. An observation can be made with regard to the two types of critical headings. For an arbitrary vehicle traveling along a downhill slope of a polygonal terrain patch where braking is possible and stability-related motion constraints exist, the range of braking headings may or may not subsume the range

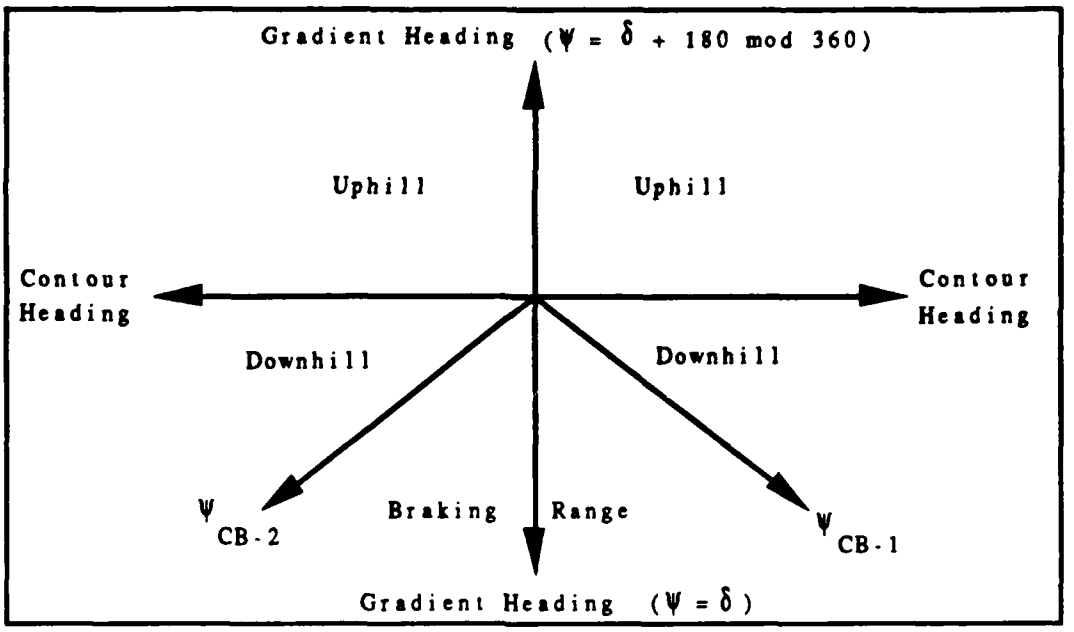

Figure 3.13 Critical Braking Headings

of (downhill) stability headings. Figure 3.14 illustrates the relationship between the critical braking azimuth and the critical stability azimuth for selected polygonal terrain patches. The motion constraints for maximum slope and stability provide the foundation for the development of a terrain classification methodology that is an essential part of the minimum-energy path-planning process.

E. SYMBOLIC TERRAIN

The components of the towed vehicle model presented in the previous sections developed a framework for describing the interaction between an arbitrary vehicle and a mathematically defined terrain surface with respect to the forces of gravity and friction. The terrain surface, represented by a polygonal tiling, consists of a set of projected polygonal terrain patches described by connected edge segments. Each projected polygon within the tiling maintains a *constant gradient* property that guarantees the uniformity of the slope and orientation of the patch within a designated threshold.

To solve the minimum-energy path-planning problem, it is beneficial to attach *symbolic* descriptions to the projected polygonal terrain patches promoting a classification based on the degree of difficulty of traversal. The degree of difficulty is attributable to surface configuration properties and surface

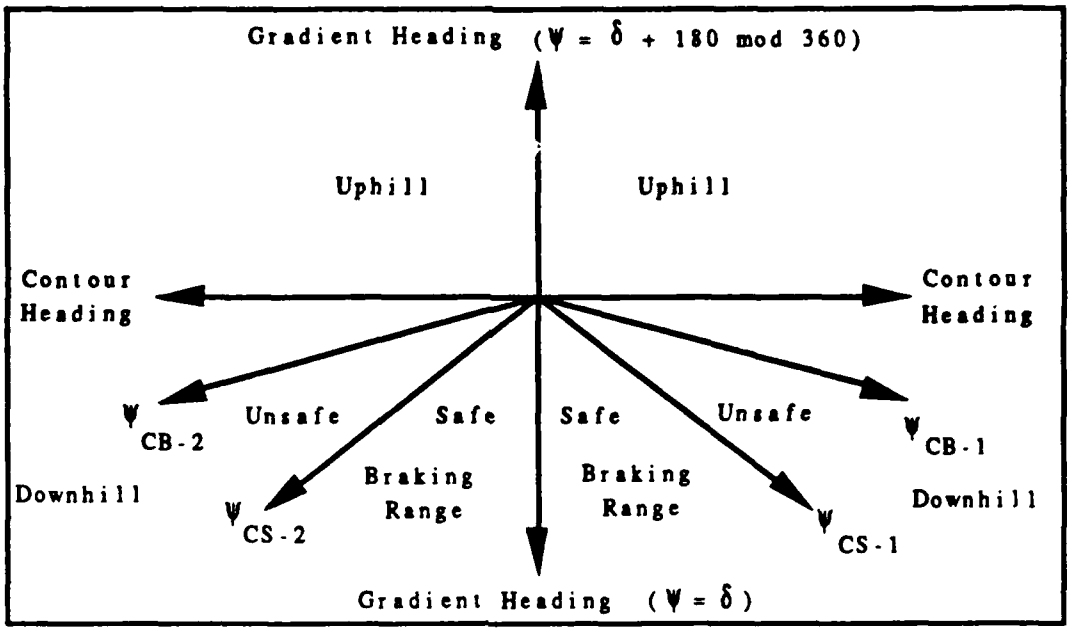

Figure 3.14 Critical Braking and Stability Headings

composition properties. Partitioning the terrain surface patches into regions of similar characteristics is an essential "preprocessing" phase in the path-planning process. A robust symbolic description of the natural terrain provides a sound problem representation that facilitates *reasoning* about large regions of the map instead of individual data points found in the gridded representations of digital terrain databases. The ability to reason about large areas of the terrain with similar properties is fundamental to the ray tracing approach proposed for planning optimal routes.

1. Taxonomy of Symbolic Terrain Surfaces

In general, the classification of natural terrain can be related to the cost of traversing the terrain. For the towed vehicle model, traversal costs are measured in terms of energy. A simple ternary classification is used to describe the degree of difficulty a vehicle encounters in attempting to negotiate a terrain surface. Thus, for a particular area of terrain, traversability with respect to energy expenditure and motion constraints can intuitively be designated as *go*, *no-go*, or *conditional-go*. This designation is in contrast to the traditional path-planning models such as [Ref. 49] that posit binary terrain using a strict go or no-go criteria with respect to obstacle and non-obstacle areas. The additional category of conditional-go results from a partitioning of non-obstacle regions by considering the concept of directional dependency

To accomplish this classification, it is necessary to employ the principles of *isotropism* and *anisotropism*. An isotropic phenomenon is one in which the relevant properties are identical in all directions. Conversely, an anisotropic phenomenon is one in which the relevant properties differ according to the direction of measurement.

By applying these principles to the towed vehicle model and the symbolic terrain, a *classification hierarchy* is developed to assist in the partitioning process. The hierarchy can be viewed as a three-level tree structure with each successive level providing additional classification information. The polygonal terrain patch, or uniform gradient region, serves as a distinguished (root) node at level zero of the tree. Each node in the hierarchy, with the exception of the root node, inherits classification parameters from its ancestors. This produces a cumulative set of restrictions to assist in the partitioning process. Figure 3.15 illustrates the general tree structure that represents the classification hierarchy.

At level one of the hierarchy, a polygonal terrain patch can be classified as an isotropic region, an anisotropic region, or as an obstacle region. In the ternary classification scheme, an isotropic region corresponds to "go" terrain, an anisotropic region corresponds to "conditional-go" terrain, and an obstacle region corresponds to "no-go" terrain. The following definitions formalize the distinction between various types of terrain in the classification hierarchy.

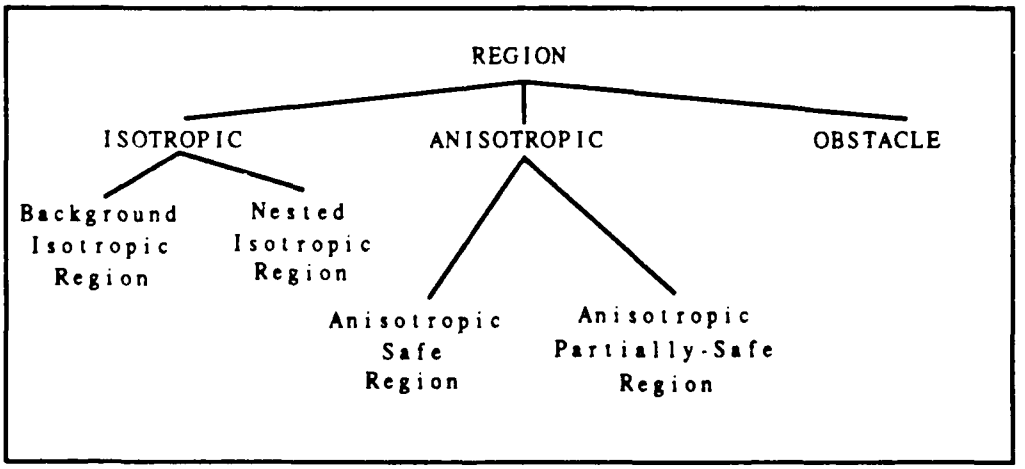

Figure 3.15 Terrain Classification Hierarchy

Definition 3.42: A polygonal terrain patch P with gradient inclination angle ϕ greater than or equal to the critical braking angle ϕ_{CB} is defined as an *obstacle region* and is expressed quantitatively as

$$\phi \geq \phi_{CB}. \tag{3.113}$$

Having isolated the regions prohibited from vehicular travel, the direction-dependent components are defined.

Definition 3.43: An arbitrary polygonal terrain patch P in which the cost of vehicle traversal per unit distance is independent of the direction of travel ψ and free from motion constraints is defined as an *isotropic region*. An isotropic region describes a polygonal terrain patch P with gradient inclination angle ϕ less than the critical coasting angle ϕ_{CC} expressed as

$$\phi < \phi_{CC}. \tag{3.114}$$

An isotropic region is distinguished by the fact that there are no motion constraints within the region and no requirement to initiate braking to keep a vehicle moving at constant speed. Motion resistance is at a minimum when a vehicle is traversing an isotropic region, that is, $\kappa = \kappa_{MIN} = \varepsilon$. Thus, an isotropic region can be interpreted as a region of maximum safety and minimum movement cost.

The other significant class of non-obstacle regions is based on the principle of anisotropism. The following definition formalizes the direction-dependent regions.

Definition 3.44: An arbitrary polygonal terrain patch P in which the cost of vehicle traversal per unit distance is dependent on the direction of travel ψ is defined as an *anisotropic region*. An anisotropic region describes a polygonal terrain patch P with gradient inclination angle ϕ greater than or equal to the critical coasting angle ϕ_{CC} and less than the critical braking angle ϕ_{CB} written as

$$\phi_{CC} \leq \phi < \phi_{CB}. \tag{3.115}$$

An anisotropic region is characterized by the absence of motion constraints for maximum slope and the presence of potential stability problems at certain vehicle azimuths. It also requires employing some form of partial braking on downhill slopes to maintain a constant speed during the traversal of the region. Motion resistance is at a level greater than the minimum resistance and less than the maximum resistance; that is, $\kappa_{MIN} < \kappa < \kappa_{MAX}$. Thus, anisotropic regions can be interpreted as regions of moderate-to-low safety and moderate-to-high movement cost.

The final level of the classification hierarchy partitions the isotropic and anisotropic regions as appropriate. Isotropic regions can be categorized as either *background* or *nested*. The following definitions formalize the two types of isotropic regions.

Definition 3.45: An arbitrary polygonal terrain patch P satisfying the criteria for isotropism, that is completely surrounded on all sides by polygonal terrain patches classified as anisotropic or obstacle, is defined as a *nested isotropic region*.

Any isotropic region not classified as nested is designated as part of the general background region.

Definition 3.46: The large, concave region satisfying the criteria of isotropism, that remains after the obstacle, anisotropic, and nested isotropic regions have been identified, represents a distinguished region defined as the *background isotropic region*.

With this definition, polygonal terrain can be viewed as a finite set of polygons superimposed on an isotropic background region. Figure 3.16 provides an illustration of the principal classes of polygonal terrain patches viewed as a polygonal tiling of the two-dimensional topographic map plane.

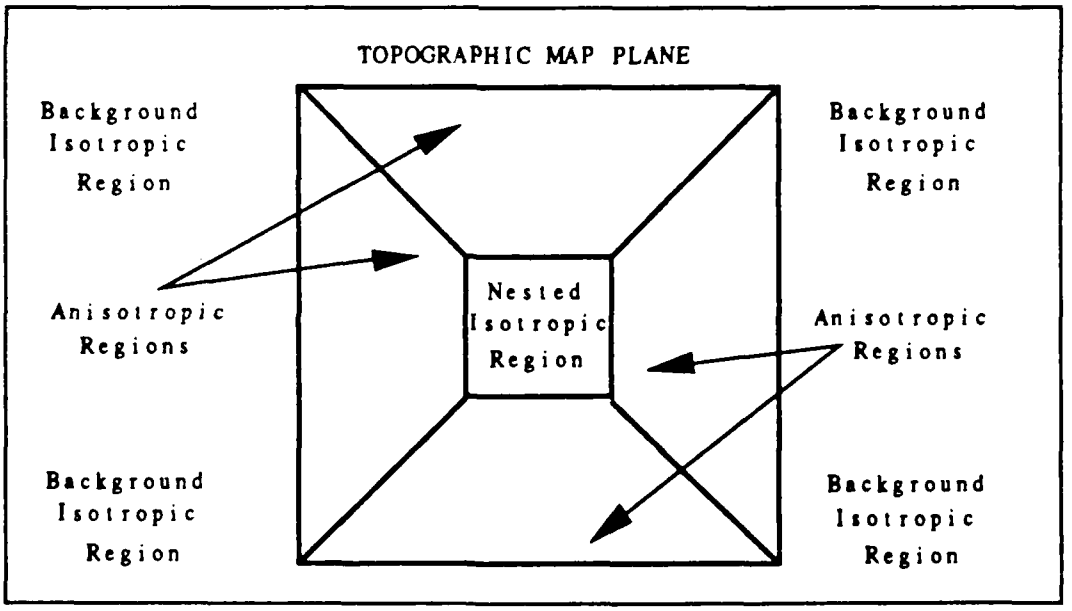

Figure 3.16 Two-dimensional Representation of Region Classes

Anisotropic regions are partitioned into two distinct categories based on the presence or absence of stability-related motion constraints. The following definitions formalize the two types of anisotropic regions.

Definition 3.47: An arbitrary polygonal terrain patch P satisfying the criteria for anisotropism, with gradient inclination angle ϕ less than the critical stability angle ϕ_{CS}, is defined as an *anisotropic-safe region* expressed quantitatively as

$$\phi_{CC} \leq \phi < \phi_{CS}. \tag{3.116}$$

An anisotropic-safe region can be viewed as an area free from stability-related motion constraints in which braking is required on downhill slopes in order to maintain constant speed within the region. Motion resistance for an anisotropic-safe region has the same limits as the resistance defined for a general anisotropic region. The presence of stability constraints within an anisotropic region provides the fundamental restriction for the final direction-dependent region classification.

Definition 3.48: An arbitrary polygonal terrain patch P satisfying the criteria for anisotropism, with gradient inclination angle ϕ greater than or equal to the critical stability angle ϕ_{CS} and less than the critical braking angle ϕ_{CB}, is defined as an *anisotropic-partially-safe region*, written as

$$\phi_{CS} \leq \phi < \phi_{CB}. \tag{3.117}$$

An anisotropic-partially-safe region can be interpreted as an area containing stability-related motion constraints in which braking is required on downhill slopes in order to maintain constant speed within the region. Motion resistance for an anisotropic-partially-safe region has the same limits as the resistance defined for a general anisotropic region.

2. Anisotropic Obstacles

Traditional mobility models described in [Ref. 23] define an obstacle region as an area of the terrain in which vehicle travel is restricted due to certain well-defined constraints that are, in essence, direction independent. An infinite traversal cost is associated with the obstacle region. Since the region is considered an "obstacle" irrespective of the vehicle azimuth ψ at which it is encountered, this region of impermissibility can be described as an *isotropic obstacle*. With the introduction of the concept of anisotropism for region classification and for the computation of traversal costs, a direction-dependent *virtual* obstacle is defined based on the stability-related motion constraints associated with a particular vehicle on a specified polygonal terrain patch.

Definition 3.49: Given an arbitrary vehicle located at position (x_p, y_p, z_p) in an anisotropic partially-safe region, the range of impermissible vehicle headings described by the set of critical stability azimuths designates a virtual obstacle area defined as an *anisotropic obstacle*.

An anisotropic obstacle does not have a static physical boundary does an isotropic obstacle. The boundary is dynamic in the sense that the virtual obstacle is a function of the current position in the region and the critical stability azimuths for that region. An anisotropic obstacle can be viewed as a wedge that fans out from the wedge tip (current vehicle location) and intersects the boundary of the polygonal terrain patch. A vehicle heading occurring within one of the designated wedges is considered an impermissible heading for stability purposes. Changing positions within the anisotropic region results in a corresponding movement of the wedges and creation of a new anisotropic obstacle. Figure 3.17 illustrates several occurrences of anisotropic obstacles within a polygonal terrain patch.

3. Homogeneous Mobility Regions

The object of the towed vehicle model and associated symbolic terrain is to develop a two-dimensional representation of the natural terrain in which the key factors in computing minimum-energy

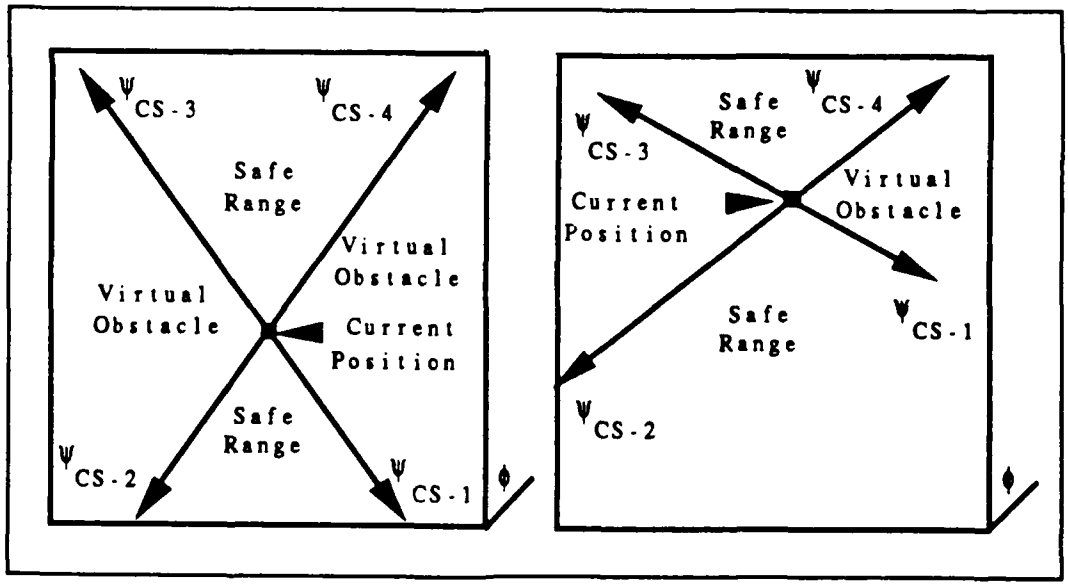

Figure 3.17 Anisotropic Obstacles

paths are a function of the vehicle motion resistance and distance traveled in the topographic map plane. The representation of the terrain as a set of polygonal terrain patches provides the initial structures that are projected into the topographic map plane. These projected polygonal terrain patches, forming the polygonal tiling of individual uniform gradient regions, can be further partitioned using other significant factors for vehicle mobility. For example, the terrain surface composition is an important factor in obtaining the vehicle motion resistance. It is possible to have multiple soil types within a single uniform gradient region. Thus, a second level of partitioning must occur to maintain the characteristics of homogeneity within the projected polygonal terrain patch.

Definition 3.50: Given an arbitrary projected polygonal terrain patch P within the topographic map plane, a contiguous region strictly contained within the patch that describes an area of uniform surface composition (soil properties) is defined as a *homogeneous mobility region.*

With this definition, the following assumption can be made regarding the properties of projected terrain patches.

Assumption 3.7: A homogeneous mobility region is a two-dimensional representation of an area of the natural terrain with constant surface configuration properties and constant surface composition properties defined within a designated threshold.

This assumption guarantees that every region on the map has a distinct *cost rate* associated with vehicle motion resistance. Therefore, for non-braking episodes, the cost of traversing a homogeneous mobility region is a function of the minimum coefficient of motion resistance and the straight-line distance traveled. For braking episodes, the cost is simply a function of the elevation difference Δh between the path entry and path exit points in the region. Figure 3.18 illustrates the concept of homogeneous mobility regions within a topographic map plane.

It should be noted, again, that the small error factor introduced by traveling along non-gradient paths is assumed to be insignificant from a global path-planning perspective. There are several reasons for this assumption. First, there are many opportunities for error in the original map data. This can be attributed to variations in the data collection and/or data recording process, changes in the environmental surroundings, or inaccuracies in the digitization process from the topographic (source) map. Second, the empirical measurement of the vehicle specific resistance is not precise. Third, a small amount of error results from the generation of symbolic terrain (polyhedron) from gridded data. Fourth, and perhaps most important is that any global path plan is subject to the local path perturbations that can occur during plan execution. Thus, the "weighting factor" in Eq. (3.108) resulting from non-gradient path traversals is

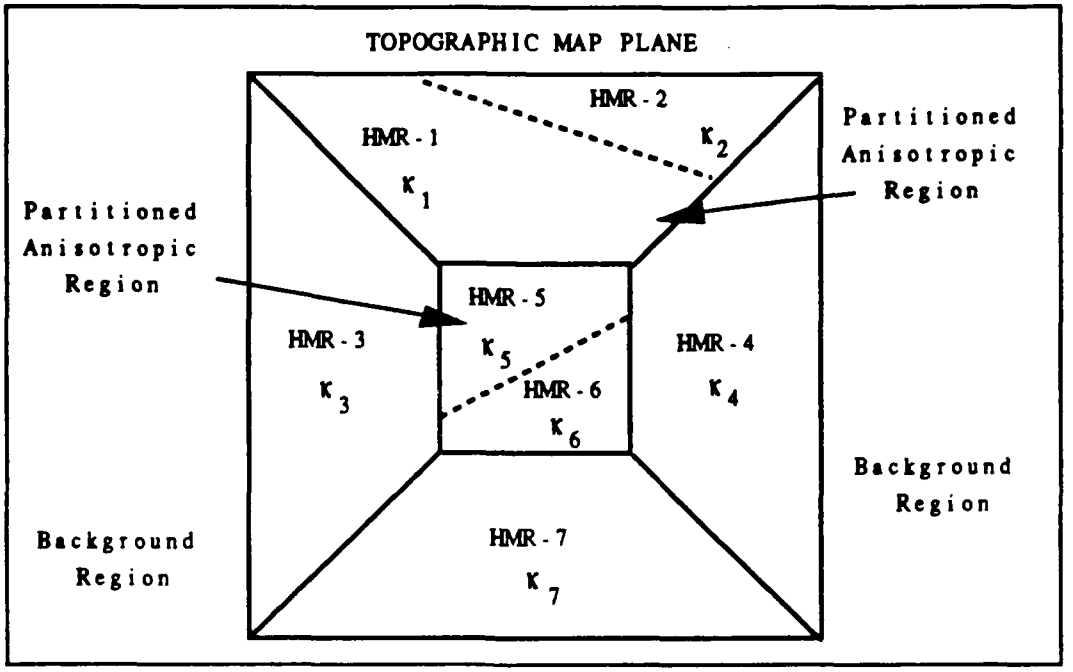

Figure 3.18 Homogeneous Mobility Regions

effectively ignored in the actual implementation discussed in Chapter IV. Without loss of generality, the traversal costs can be classified as braking costs and non-braking costs. This conceptual simplicity is important because the cost of every optimal path traversal is *heading independent* unless braking is involved.

From a theoretical perspective, however, it is recognized that the actual cost resulting from non-gradient path traversals can be computed mathematically using Eq. (3.108). The non-braking traversals are *heading dependent* in this case, and the costs include the slope of each terrain patch and vehicle heading inclination angle on that patch. Ignoring the error factor from the "cosine effect" results in a small but consistent overestimate of the minimum-energy traversal cost between any two points on the natural terrain.

F. SUMMARY

A mathematical model has been proposed to predict the energy requirements of vehicles operating in an off-road environment in natural terrain. The terrain is modelled as a set of polygonal regions with each constituent polygonal terrain surface patch representing a uniform gradient region. A classification

hierarchy partitions the uniform gradient regions into three fundamental categories based on the capability of the vehicle to negotiate the particular soil-slope combination. Further subdivisions are possible due to the principles of isotropism and anisotropism which introduce the concept of direction dependency in computing traversal costs.

The principal focus of the towed vehicle model is in the separation of resistance energy costs involving Coulomb friction forces from the potential-energy costs associated with the force of gravity. For the global, minimum-energy path-planning problem, potential energy can be factored out as a constant term and kinetic energy is ignored. Thus, all remaining costs are resistive in nature and a function of the vehicle motion resistance and the straight-line map distance. The distinction between the "horizontal" and "vertical" components of energy cost facilitates the development of a two-dimensional path-planning model that utilizes three-dimensional information for motion constraints. This is achieved by defining all costs, constraints, and distances on the two-dimensional projection of the three-dimensional surface representing the natural terrain. The two-dimensional problem representation is conceptually simpler with respect to any eventual search process that attempts to find minimum-energy paths. It also corresponds more closely to the topographic maps traditionally used for route planning applications.

IV. OPTIMAL-PATH-PLANNING ALGORITHM

A. INTRODUCTION

The planning of optimal paths through natural terrain is fundamentally a search problem. The solution to a path-planning problem requires a suitable *problem representation* capturing the relevant characteristics of the particular vehicle in transit as well as the key features of the surrounding environment. In addition, there must be an effective *strategy* for conducting the search. The preceding chapter introduced a mathematical model describing the energy costs of vehicular travel across natural terrain surfaces. Determining the most fuel-efficient route between any two points on the map necessitates the integration of the theoretical concepts developed in the towed vehicle model with an effective path-planning algorithm.

In this dissertation, the minimum-energy path-planning problem is formulated as a state-space representation consistent with the approach of Nilsson [Ref. 1]. The state-space formulation has three basic components: (1) a description of *states* within the state space, (2) a specification of *operators*, or successor functions providing transitions between states, and (3) a *control strategy* establishing the precedence among operators during the search. The following sections discuss in detail the composition of states, the various pruning criteria utilized to reduce the size of the search, the behavior of optimal paths in isotropic and anisotropic regions, and the algorithm developed to compute minimum-energy paths.

B. PROBLEM REPRESENTATION

1. Windows and Regions

The optimal-path-planning algorithm developed in this dissertation requires a geometric description of the natural terrain, a vehicle concept describing the key attributes of the prime mover, and a mission statement asserting global start and goal positions. The required geometric description of the terrain consists of a finite set of vertices V and a finite set of edges E forming a convex-polygonal tiling of the two-dimensional topographic map plane. The tiling results in a set of convex polygons corresponding to

the homogeneous mobility regions described in the previous chapter.‡ These polygons will be designated as *search regions*. The attributes pertaining to the slope and orientation of the region, i.e., configuration information, are strictly terrain-dependent. The remaining attributes specifying the isotropic, anisotropic, or obstacle classification as well as the stability and braking constraints depend on both the vehicle and the terrain.

Given the set of search regions R, it is possible to identify the set of geometric entities an optimal path must pass through enroute from the start position to the goal position. Referring to Figure 4.1, a formal definition is provided.

Definition 4.1: The finite set of vertices V and the finite set of edges E forming a polygonal tiling of the two-dimensional topographic map plane represent the set of *search windows*. Each search window W_k can be classified as a *vertex window* or an *edge window*. A particular set of vertex windows and edge windows denoted as *boundary windows* form an irregular convex polygon and bound the optimal path.

An edge window borders two search regions, R_i and R_j, one designated as the *pre-frontier* search region

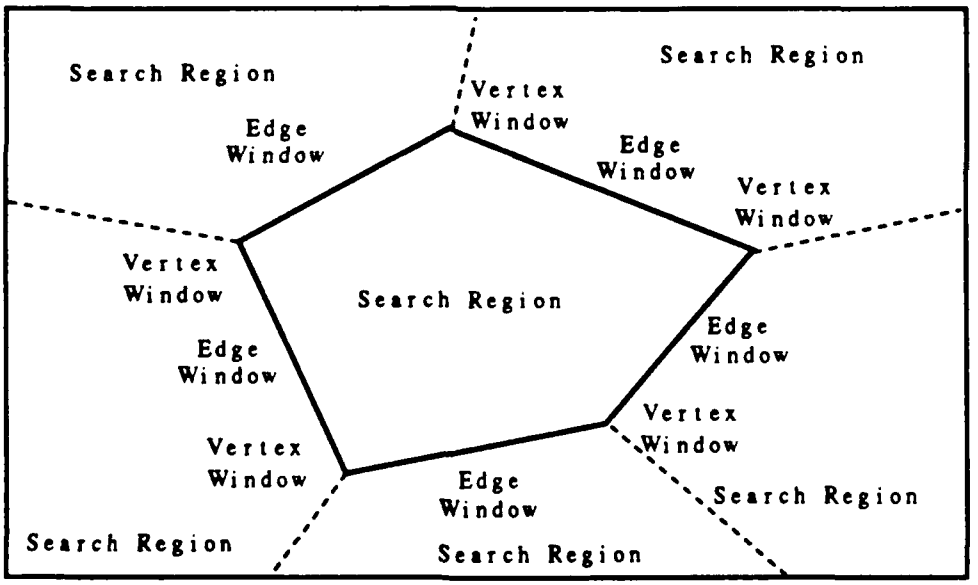

Figure 4.1 Search Windows and Search Regions

‡ For purposes of the minimum-energy path-planning algorithm, it is assumed the coefficient of specific resistance ε is uniform across the terrain map.

and the other as the *post-frontier* search region depending on direction of window expansion, as will be explained. Similarly, a vertex window borders a finite set of search regions, one of which is identified as the pre-frontier search region with the remaining search regions categorized as post-frontier search regions. Bounds on the optimal path can be established by the physical limits of the terrain map or by a bounding ellipse as in [Ref. 2].

A search over the space of all possible paths between the start and the goal solves the minimum-energy path-planning problem. Thus, it is important to use any information about the vehicle or the terrain that can serve to reduce the number of paths considered in the search process. Two very powerful methods employed to simplify the search are *geometric visibility analysis* and *vehicle heading analysis*.

2. Geometric Visibility Analysis

Visibility analysis provides the initial screening of map locations that can be directly reached by the vehicle from a given other location. The polygonal tiling of the terrain map implies a concise visibility graph with respect to vertices and edges. This is due to the connectivity properties of convex polygons. A vertex window is "visible" from another vertex window if the windows are members of the same search region. Similarly, visibility exists between two edge windows if both windows are part of the same region. Membership in the same region is necessary but not sufficient to establish visibility between a vertex window and an edge window. In addition, windows may not overlap. Therefore, endpoint-vertex windows associated with an edge window are not visible from that edge window.

3. Vehicle Heading Analysis

A significant factor that must be considered in moving from search window to search window is the range of permissible headings. Vehicle heading analysis is used in the optimal-path-planning algorithm to: (1) eliminate search regions (isotropic and anisotropic obstacles), and (2) reduce the area considered within the remaining search regions. Three classes of "critical headings", i.e., geometric, stability, and braking, are required for the analysis. Specific permissible heading ranges arise for each of these classes.

Geometric headings refer to the configuration of the convex polygons on the two-dimensional map plane. Let W_i and W_{i+1} be intervisible search windows with W_i the start window. A heading generated by connecting a pair of endpoints, one from W_i and one from W_{i+1}, is a critical geometric heading ψ_{CG}, $0 \leq \psi_{CG} < 360$. For W_i and W_{i+1} edge windows, four critical geometric headings are

generated between windows. The range of permissible headings in going by straight line from a point on W_i to a point on W_{i+1} is bounded by the two endpoint-connection segments that intersect, or the "cross headings", and represents a geometric heading range H_{GM}. A heading range can be treated as an "open" or a "closed" interval depending on whether the endpoint-connection segments are included in the range or not. An endpoint-connectio. segment from one vertex window to another is tagged as a closed heading interval. Otherwise the interval is open.

The stability-heading ranges establish the permissible headings for stable optimal path traversals between two search windows, that is, the heading range for stability H_{ST}. Critical stability headings can be defined according to Definitions 3.38 and 3.39. From Figure 3.12, note that the four critical stability headings partition the set of all possible headings into four distinct ranges. ψ_{CS-1} and ψ_{CS-2} bound the range of headings a vehicle can safely traverse on a "downhill" slope without catastrophic overturn; ψ_{CS-3} and ψ_{CS-4} bound the range for an uphill slope. Both heading ranges are closed intervals. The single-heading, non-braking (degenerate) ranges, defined by each of the critical stability headings above, represent heading ranges for critical stability H_{CS} and are discussed in detail in Section IV.C.2. Unstable ranges are obtained trivially as the complement of the two stability ranges. Each range represents a heading range for instability H_{IN} and is an open interval.

The braking heading range defines the permissible headings requiring vehicle braking to maintain constant velocity, that is, the heading range for braking H_{BR}. The critical braking headings ψ_{CB-1} and ψ_{CB-2} can be defined according to Definition 3.40. The complement of the braking range defines the non-braking headings or the heading range for non-braking H_{NB}.

The applicable heading ranges, once derived, must be intersected to obtain a total permissible heading range $HP_{<DESIGNATION>}$ expressed as

$$HP_{<DESIGNATION>} = H_{GM} \cap H_{ST} \cap H_{<RANGE-TYPE>}. \tag{4.1}$$

The $<RANGE-TYPE>$ is either BR (braking), NB (non-braking), or CS (critical stability). The subscript $<DESIGNATION>$ indicates what adjacent-window pair this heading range describes. It is possible to produce two disjoint heading ranges as a result of the intersections. If this occurs, the ranges are treated as separate options.

Given the "universe" of all headings HU, $0 \le HU < 360$, an impermissible heading range $HI_{<DESIGNATION>}$, defined as the complement of the range of permissible headings, is expressed as

$$HI_{<DESIGNATION>} = HU - HP_{<DESIGNATION>}. \tag{4.2}$$

The heading ranges defined by Eqs. (4.1) and (4.2) partition the space of all headings into traversals that the vehicle can successfully execute and those that cannot be completed safely.

C. PROPERTIES OF OPTIMAL PATHS

1. Turn Criteria

Planning an optimal path between start and goal locations may necessitate crossing various types of search regions and search windows. An optimal path can turn either within a search region (intra-region turn) or at a boundary crossing (inter-region turn). The restrictions on turns in optimal paths, the *local turn criteria*, are paramount to the development of the transition operators (successor functions), for the optimal-path search, as discussed in Section IV.E.3.

Lemma 4.1: If OP is an optimal path between a given start window S_V and goal window G_V, then the path between any two points on OP must also be optimal.

Proof: The proof is by contradiction. Consider Figure 4.2. Let OP represent an optimal path from start window S_V to goal window G_V passing through points P_1 and P_2. Assume there is some path between P_1 and P_2 passing through point Q such that the cost of the path from P_1–Q–P_2 is less than the cost of the

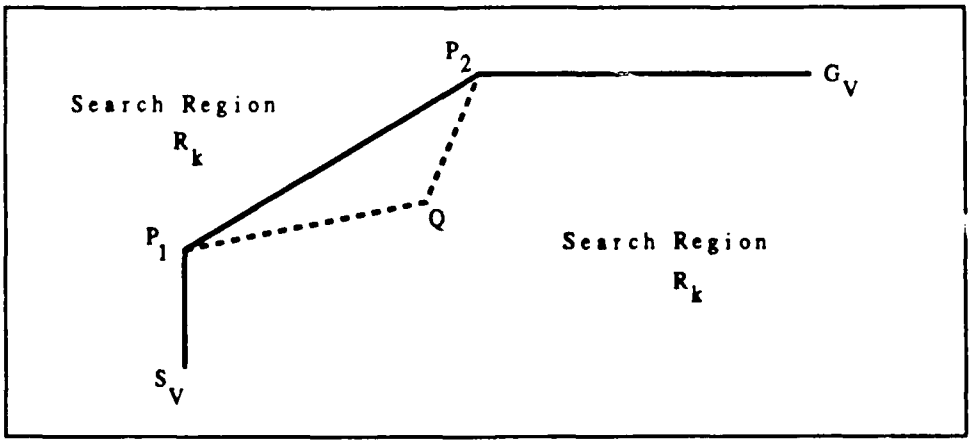

Figure 4.2 Optimal Path Segments

path segment P_1–P_2 along the original path. Then, the path S_V–P_1–Q–P_2–G_V must have a lower cost than OP. But OP is the optimal path between S_V and G_V and therefore, a path P_1–Q–P_2 having a lower cost than the path segment P_1–P_2 cannot exist.

QED.

A fundamental theorem describes the intra-region behavior of an optimal path.

Theorem 4.1: If optimal path OP emanates from frontier search window W_k enroute to post-frontier search window W_{k+1} crossing search region R_k, then the heading (azimuth) ψ of the path OP must remain constant within R_k unless turns are executed across a stability-impermissible range from one critical stability heading to another critical stability heading.

Proof: The proof is by induction and has two parts. Let n represent the number of turns within the search region.

Part 1 (Basis): The 1-Turn Path

It must be shown that for an optimal path crossing a search region at a given heading, it is never advantageous (energy costwise) to execute a single turn within the region. Refer to Figure 4.3. Let P_a and P_b denote points on search windows W_k and W_{k+1}, respectively, representing the entry and exit points of an optimal path OP within search region R_k. Let P_c represent an arbitrary point within the search region. Let OP_1 denote a path consisting of a single "permissible" path segment PE_1:(P_a, P_b) and let OP_2 denote a path consisting of two "permissible" path segments PE_2:(P_a, P_c) and PE_3:(P_c, P_b). Let D_1, D_2, and D_3

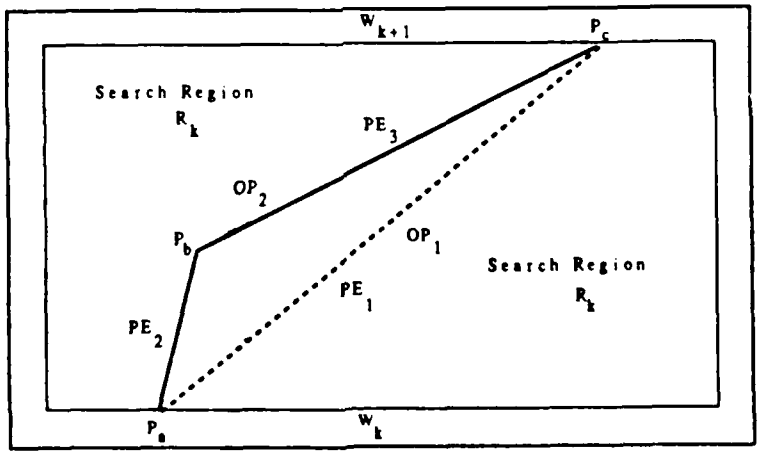

Figure 4.3 Single-turn Path Behavior

represent the lengths of the respective path segments and let θ_1, θ_2, and θ_3 denote the heading inclination angles of the respective segments. The differences in elevations of the points P_a, P_b, and P_c, can be expressed by

$$\Delta h_1 = D_1 \tan\theta_1, \tag{4.3a}$$

$$\Delta h_2 = D_2 \tan\theta_2, \tag{4.3b}$$

$$\Delta h_3 = D_3 \tan\theta_3. \tag{4.3c}$$

By geometry,

$$\Delta h_1 = \Delta h_2 + \Delta h_3. \tag{4.4}$$

Using Eq. (3.77), for the single path segment OP_1 the traversal cost of a non-braking episode is expressed as

$$U_{OP-1} = \varepsilon mgD_1. \tag{4.5a}$$

The equivalent traversal cost for the single path segment braking episode is expressed as

$$U_{OP-1} = mg\,\Delta h_1. \tag{4.5b}$$

Basis Case 1a: Single Segment Non-Braking/Multiple Segment Non-Braking

Using Eq. (3.46), the traversal cost U_{OP-1} can be rewritten as

$$U_{OP-1} = mgD_1 \tan\theta_{CC}. \tag{4.6a}$$

The traversal cost U_{OP-2} can be expressed as

$$U_{OP-2} = mg\,(D_2 + D_3)\tan\theta_{CC}. \tag{4.6b}$$

Since $D_1 < D_2 + D_3$, it is evident that

$$mgD_1 \tan\theta_{CC} < mg\,(D_2 + D_3)\tan\theta_{CC}, \tag{4.7}$$

and the straight-line path OP_1 generates a lower cost.

Basis Case 1b: Single Segment Braking/Multiple Segment Braking

Using Eq. (3.46), the traversal cost U_{OP-1} can be rewritten as

$$U_{OP-1} = mgD_1 \tan\theta_1. \tag{4.8a}$$

The traversal cost U_{OP-2} can be expressed as

$$U_{OP-2} = mg\,(D_2\tan\theta_2 + D_3\tan\theta_3). \tag{4.8b}$$

Since $\Delta h_1 = \Delta h_2 + \Delta h_3$, the traversal cost U_{OP-1} is equal to the traversal cost U_{OP-2}. Considering the second-order cost involved in initiating a turn, the straight-line path must have a lower traversal cost.

Basis Case 1c: Single Segment Braking/Multiple Segment Non-Braking

Refer to Figure 4.4. Let OP_1 represent a braking path and OP_2 a non-braking "switchback" path with path segments defined as above. Let OP_3 represent a path consisting of two segments PE_4 and PE_5 such that the two path segments cross region R_k at the critical braking headings and form a second "switchback" that lies within the bend of path OP_2. Let D_4 and D_5 represent the lengths of PE_4 and PE_5 and let θ_4 and θ_5 be the respective heading inclination angles. The traversal cost of U_{OP-1} can be expressed as

$$U_{OP-1} = mgD_1\tan\theta_1 \tag{4.9a}$$

and the non-braking traversal cost of U_{OP-2} is written as

$$U_{OP-2} = mg\,(D_2 + D_3)\tan\theta_{CC}. \tag{4.9b}$$

The traversal cost of U_{OP-3} can be expressed as

$$U_{OP-3} = mg\,(D_4 + D_5)\tan\theta_{CC}. \tag{4.9c}$$

Since the path OP_3 turns "inside" of path OP_2, it is evident that $(D_4 + D_5) < (D_2 + D_3)$. Thus, the traversal

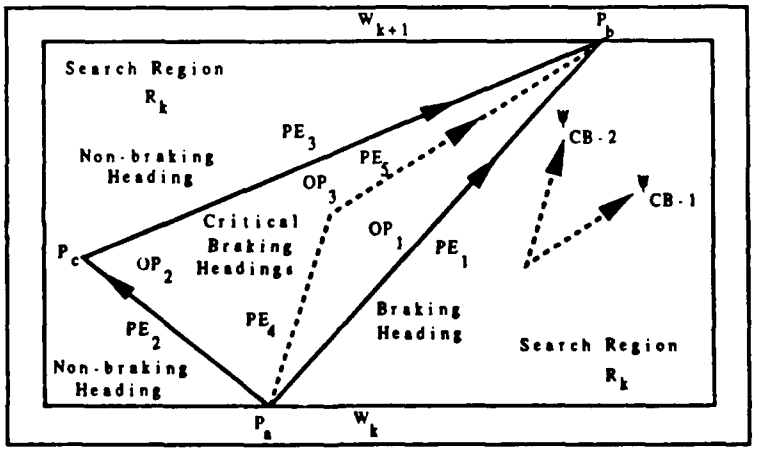

Figure 4.4 Non-braking Switchbacks

cost of OP_3 must be less than the traversal cost of OP_2. But, at the critical braking heading, the traversal cost U_{OP-3} is the same whether the path OP_3 is considered as a braking or non-braking path. If it is interpreted as braking, then by Basis Case 1b, U_{OP-3} is equal (in first-order cost) to U_{OP-1}. Since $U_{OP-3} < U_{OP-2}$ and $U_{OP-3} = U_{OP-1}$, it must be the case that $U_{OP-1} < U_{OP-2}$; that is, the straight-line (braking) path must have a lower traversal cost.

Basis Case 1d: Single Segment Non-Braking/Multiple Segment Braking

Using Eq. (3.46), the traversal cost U_{OP-1}, can be rewritten as

$$U_{OP-1} = mgD_1\tan\theta_{CC}. \tag{4.10a}$$

The traversal cost U_{OP-2}, can be expressed as

$$U_{OP-2} = mg\,(D_2\tan\theta_2 + D_3\tan\theta_3). \tag{4.10b}$$

Since both path segments in OP_2 are braking episodes, $\theta_1 > \theta_{CC}$ and $\theta_2 > \theta_{CC}$. It is also the case that $D_1 < D_2 + D_3$. Thus, it is evident that

$$mgD_1\tan\theta_{CC} < mg\,(D_2\tan\theta_2 + D_3\tan\theta_2). \tag{4.11}$$

and the straight-line path OP_1 generates a lower traversal cost.

Basis Case 1e: Single Segment Braking/Multiple Segment Hybrid

Using Eq. (3.46), the traversal cost U_{OP-1} can be rewritten as

$$U_{OP-1} = mgD_1\tan\theta_1, \tag{4.12a}$$

and the traversal cost U_{OP-2} can be expressed as

$$U_{OP-2} = mg\,(D_2\tan\theta_{CC} + D_3\tan\theta_3). \tag{4.12b}$$

From Basis Case 1c, it is true that $mgD_1\tan\theta_1 \le mg\,(D_2\tan\theta_{CC} + D_3\tan\theta_{CC})$. Since path segment PE_3 is a braking episode, $\theta_3 > \theta_{CC}$. Therefore, it must be true that

$$mgD_1\tan\theta_1 < mg\,(D_2\tan\theta_{CC} + D_3\tan\theta_3), \tag{4.13}$$

and the straight-line path OP_1 generates a lower traversal cost.

Basis Case 1f: Single Segment Non-Braking/Multiple Segment Hybrid

With Eq. (3.46), the traversal cost U_{OP-1}, can be rewritten as

$$U_{OP-1} = mgD_1\tan\theta_{CC}. \tag{4.14a}$$

The traversal cost U_{OP-2} can be expressed as

$$U_{OP-2} = mg\,(D_2\tan\theta_{CC} + D_3\tan\theta_3). \tag{4.14b}$$

From Eq. (4.13), it is evident that $mgD_1\tan\theta_1 < mg\,(D_2\tan\theta_{CC} + D_3\tan\theta_3)$. Since path segment PE_1 is a non-braking episode, $\theta_{CC} < \theta_1$. Thus, it is obvious that

$$mgD_1\tan\theta_{CC} < mg\,(D_2\tan\theta_{CC} + D_3\tan\theta_3), \tag{4.15}$$

and the straight-line path OP_1 generates a lower traversal cost.

Part 2 (Induction): The n-Turn Path

It must be shown that if it is never advantageous (costwise) for an optimal path to turn n times within a search region then it is never advantageous (costwise) for the same path to turn $n+1$ times. The inductive hypothesis is that it is never advantageous for an optimal path to turn n times within a region. Consider Figure 4.5. Let OP_1 represent an $(n+1)$-*turn* path bounded by endpoint vertices P_a and P_c. Let OP_2 denote an n-*turn* path representing that part of OP_1 bounded by endpoint vertices P_a and P_b. Assume OP_1 is an optimal path from P_a to P_c. Thus, it *is* advantageous to turn $n+1$ times within a region. By Lemma 4.1, since OP_1 is an optimal path, then OP_2 must also be an optimal path. But OP_2 cannot be optimal and turn n times (violates the inductive hypothesis). By contradiction, OP_1 cannot be an optimal path, and therefore, it is not advantageous for an optimal path to turn $n+1$ times within a region given that it is not advantageous for the path to turn n times.

QED.

Corollary 4.1: An optimal path cannot follow a *curved* path within a search region.

From the calculus, an optimal path consisting of k straight-line path segments and $k-1$ turns within the region can, in the limit as $k \to \infty$, represent a curved path. From the proof of Theorem 4.1, it is never advantageous (cost-wise) to turn n times within a region. Thus, the curved path cannot be the optimal path.

2. Traversal Types

Since each search region is classified according to the type of constraints that can occur within its boundaries, a set of *traversal types* can be specified for optimal paths within each type of region. It is shown in this dissertation that an optimal path can traverse a region in one of only four ways: (1) straight across at a non-braking heading, (2) straight across at a critical stability heading, (3) alternating episodes at a matched pair of critical stability headings, and (4) straight across at a braking heading [Ref. 3]. The second, third, and fourth traversal types relate to anisotropic regions only; that is, regions that have stability

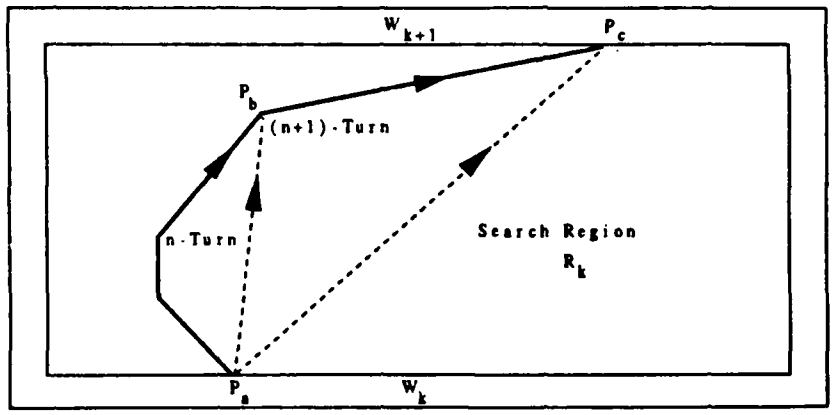

Figure 4.5 Multiple-turn Path Behavior

and/or braking constraints. A definition is provided for each of the four traversal types followed by a theorem regarding optimal path behavior. The term "path segment" is interpreted in accordance with Definition 3.25.

Definition 4.2a: A path segment *PE* that travels across a region at a constant, non-braking, non-critical-stability heading is an *unconstrained traversal* or a path traversal of *type-I*.

Since type-I traversals are non-braking episodes, the cost is a function of the minimum coefficient of motion resistance, ($\kappa = \varepsilon$) and the straight-line distance D. Henceforth, the minimum coefficient of motion resistance is designated as the *optimal cost rate*, and the associated cost for a non-braking episode is defined as the *non-braking cost*.

Definition 4.2b: A path segment *PE* that travels across an anisotropic-partially-safe search region at a constant, non-braking, critical stability heading ψ_{CS} is a *stability traversal* or a path traversal of *type-II*.

Type-II traversals have a cost computed at the optimal cost rate. The third traversal type allows for intra-region turns.

Definition 4.2c: A path segment *PE* that travels across an anisotropic-partially-safe search region alternating episodes at a matched pair of critical stability headings (either ψ_{CS-2} and ψ_{CS-3} or ψ_{CS-4} and ψ_{CS-1} in Figure 3.12), is a *discontinuous stability traversal* or a path traversal of *type-III*.

As with type-I and type-II traversals, type-III traversals have a cost computed at the optimal cost rate (for each turn segment) with a small second-order cost associated with each intra-region turn. The final traversal type is the only one involving vehicle braking.

Definition 4.2d: A path segment *PE* that travels across an anisotropic-safe or anisotropic-partially-safe search region at a constant braking heading ψ_{BR} is a *braking traversal* or a path traversal of *type-IV*.

In type-IV traversals, the traversal cost is a function of the elevation difference between the entry point and exit point of the path in the search region as discussed in Section III.B.4. The cost associated with braking episodes is the *braking cost*.

Assumption 4.1: Discontinuous stability traversals, or switchback traversals within search regions will not be considered for optimal paths.

The restriction on traversals by Assumption 4.1 exists because the prime mover (tracked or wheeled vehicle) may pass through a range of impermissible headings as it moves from one critical stability heading to the next within a particular region in discontinuous stability traversals. This situation can occur, for example, when the vehicle is moving up a slope at a critical stability heading and then turns to a downhill critical stability heading. From the mathematical model of vehicle-terrain interaction developed in Chapter III, the vehicle is assumed to move at a low, constant speed and cannot rely on the effects of centrifugal force to counteract a potentially unstable position. Therefore, traversing an impermissible heading increases the probability of a stability failure, i.e., catastrophic overturn.

Theorem 4.2: If optimal path *OP* emanates from frontier search window W_k enroute to post-frontier search window W_{k+1} crossing search region R_k, and is prohibited from executing discontinuous stability traversals, then *OP* must traverse the homogeneous region with a *type-I*, *type-II*, or *type-IV* traversal.

Proof: By Theorem 4.1 and Assumption 4.1, the heading of an optimal path must remain constant within a search region. Consider Figure 4.6. The space of all headings can be described using a tree. From the mathematical model in Chapter III, an optimal path can cross a region at either a non-braking or braking heading. Thus, at level one of the tree, the space of all headings can be partitioned into two classes. Braking headings are *type-IV* traversals. The non-braking headings can be further partitioned into headings that are impermissible in the region, headings that lie along the boundary of the permissible and impermissible ranges (critical stability headings, type-II), and all other headings (which must be permissible but not critical, hence type-I).

QED.

To complete the analysis of optimal path behavior, isotropic obstacles must be considered. An optimal path that reaches an edge window of an isotropic obstacle region terminates, whereas an optimal path that intersects an obstacle vertex window can be subject to certain kinds of further expansion. Table 4.1 provides a summary of path traversal types, associated cost factors, and permissible region types.

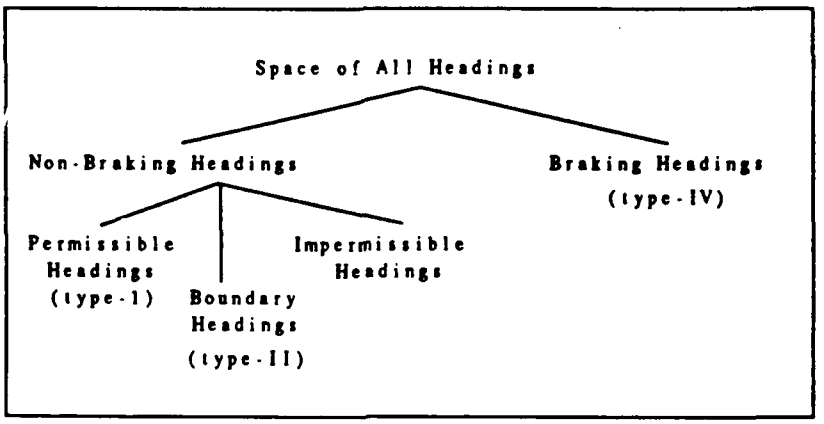

Figure 4.6 Path Heading Space

D. REGION-BOUNDARY CONSTRAINTS

To confirm the validity of moving from one window to another on a path requires the application of a set of region-boundary (RB) constraints. The *RB* constraints are classified according to the types (I, II, or IV) of path traversals in the two regions adjacent the window. The two heading ranges produced as a result of applying the boundary constraints between the two regions are the region-boundary-constraint heading ranges $HC_{<DESIGNATION>}$. The possible values for *<DESIGNATION>* are structured window triples. In the following equations, $HC_{(1)23}$ indicates the range of headings for traversal from window 2 to window 3, based on the heading range from window 1 to window 2; $HC_{12(3)}$ describes the range of headings from window 1 to window 2, given a heading range from window 2 to window 3. This means that the pre-frontier heading range affects the post-frontier heading range. It also suggests "backward reasoning" about the heading range in the pre-frontier region based on the post-frontier region, the technique "constraint propagation" used for solving certain artificial-intelligence problems. Thus, each type of *RB* constraint has

Table 4.1 OPTIMAL PATH BEHAVIOR

Traversal Type	Cost Factor	Permissible Region Types (See Section III.E.1)
Type-I	Non-braking	Isotropic Anisotropic-safe Anisotropic-partially-safe
Type-II	Non-braking	Anisotropic-partially-safe
Type-IV	Braking	Anisotropic-safe Anisotropic-partially-safe

two heading-range-update formulas associated with it: one for the post-frontier region and one for the "revised" pre-frontier region.

The analysis of the region-boundary constraints can be simplified because of the symmetry of the constraint pairs. Given a *type-X* to *type-Y* traversal pair, such that $X, Y \in \{I, II, IV\}$, the same path behavior is exhibited for a *type-Y* to *type-X* traversal pair. This can be shown informally by reversing the direction of the gradient on the terrain patch and traversing the path in the opposite direction. Thus, the total number of region-boundary constraints can be reduced from twelve to six. Table 4.2 provides a summary of the *RB* constraints.

1. I-I RB Constraints

The first RB constraint involves two successive unconstrained traversals. Figure 4.7 illustrates a type I-I region-boundary constraint RBC_{I-I}. The following theorem describes the behavior of an optimal path for RBC_{I-I}.

Theorem 4.3 If optimal path OP executes consecutive type-I traversals across regions R_k and R_{k+1} respectively, then the path cannot turn on the boundary W_k between the regions.

Proof: Let $PE_1:(P_a, P_b)$ represent the straight-line path OP_1 formed by two consecutive type-I traversals.

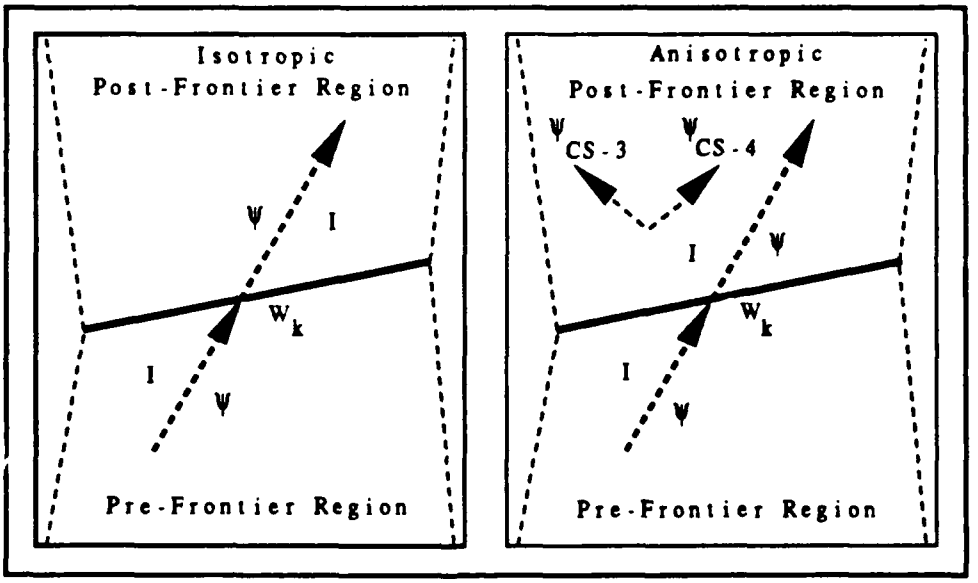

Figure 4.7 Type I-I Region-boundary Constraint

Let PE_2:(P_a, P_c) and PE_3:(P_b, P_c) represent the path OP_2 formed by two consecutive type-I traversals that turn at the boundary crossing. By Definition 4.2a, type-I traversals must be non-braking episodes. Therefore, by Basis Case 1a, Theorem 4.1, OP_1 must have a lower cost than OP_2 since distance is the minimization criteria. Thus, the consecutive type-I traversals cannot turn on the boundary W_k.

QED.

The constraint dictates that an optimal path not turn when crossing the boundary between regions. Hence, the heading ranges must be identical, so

$$HC_{(1)23} = HP_{12} \cap HP_{23} = HC_{12(3)}. \tag{4.16}$$

2. I-II RB Constraints

Figure 4.8 illustrates a type I-II region-boundary constraint RBC_{I-II}. The following theorem describes the behavior of an optimal path for RBC_{I-II}.

Theorem 4.4: If optimal path OP executes a type-II traversal in region R_{k+1} and a type-I traversal in an adjacent region R_k, then the heading ψ associated with the type-I traversal must be impermissible in region R_{k+1}.

Proof: Refer to Figure 4.8. The proof is by contradiction. Assume the type-I traversal at heading ψ in region R_k *is* permissible in the type-II region R_{k+1}. Thus, the optimal path is represented by the path $P-R-Q$ such that each segment within the path is a non-braking episode. Let S be a point infinitesimally close to point R on the edge window W_k such that the path $P-S-Q$ turns "inside" optimal path $P-R-Q$

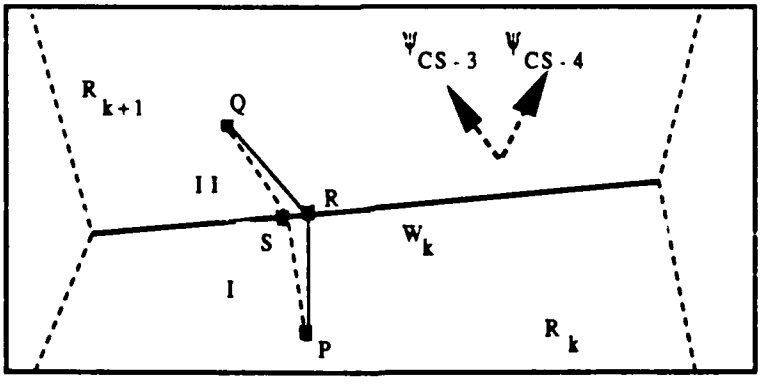

Figure 4.8 Path Turns Involving Type-I and Type-II Traversals

and consists of non-braking episodes $P-S$ and $S-Q$. By geometry, path segment $S-Q$ must be permissible in region R_{k+1} and path segement $P-S$ must be permissible in region R_k. Since all path segments are non-braking episodes, distance is the criteria for optimality. By geometry, the path $P-S-Q$ must be shorter (and less costly) than path $P-R-Q$. Therefore, path $P-R-Q$ cannot be the optimal path and the type-I traversal at heading ψ in region R_k must be *impermissible* in the type-II region R_{k+1}.

QED.

Thus, the heading ranges are written as

$$HC_{(1)23} = HP_{23} \tag{4.17a}$$

and,

$$HC_{12(3)} = HI_{23} \cap HP_{12}. \tag{4.17b}$$

3. I-IV RB Constraints

The following theorem describes the behavior of an optimal path for a type I-IV region-boundary constraint RBC_{I-IV}.

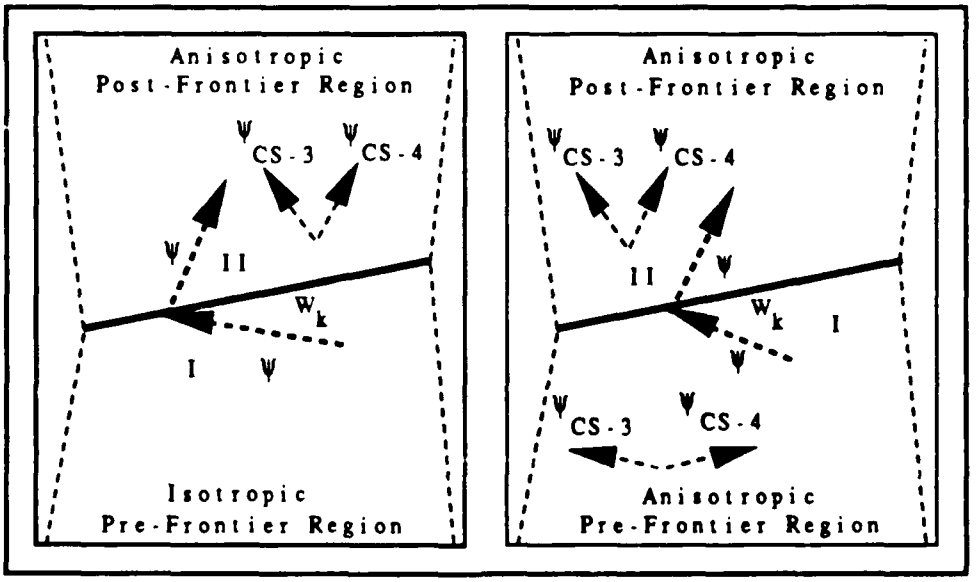

Figure 4.9 Type I-II Region-boundary Constraint

Theorem 4.5: Let ϕ represent the slope of the frontier search window with respect to the topographic map plane and ε denote the optimal cost rate. If the optimal path OP approaches edge window W_i with endpoint-vertex windows W_j and W_k across pre-frontier search region R_k at a braking heading ψ_{BR}, then, either there exists a single, non-braking exit heading $HUB_{i-1,i} = \sin^{-1}(-\tan\phi/\varepsilon)$ with respect to the boundary normal for the optimal path OP in the post-frontier region or the optimal path is constrained to pass through one of the endpoint-vertex windows W_j or W_k.

Proof: Refer to Figure 4.10 for the following derivation. Let frontier search window W_i:(W_j,W_k) participate in search regions R_k and R_{k+1}. Let Q and R represent the start point and termination point for optimal path OP crossing edge window W_i at point S. Let the path segment from R to S be a non-braking episode and the path segment from S to Q be a braking episode. Let PE represent a line segment formed by the projection of R onto edge window W_i with point T designated at the intersection point. Let D_1, D_2, and D_3 represent the distances between points R and T, T and S, and R and S, respectively. Let the elevation of points T, S, and Q be represented by h_0, h_1, and h_2. Let θ represent the angle subtended from the path segment PE to the segment of the optimal path bounded by points R and S, and let ϕ represent the slope of edge window W_i.

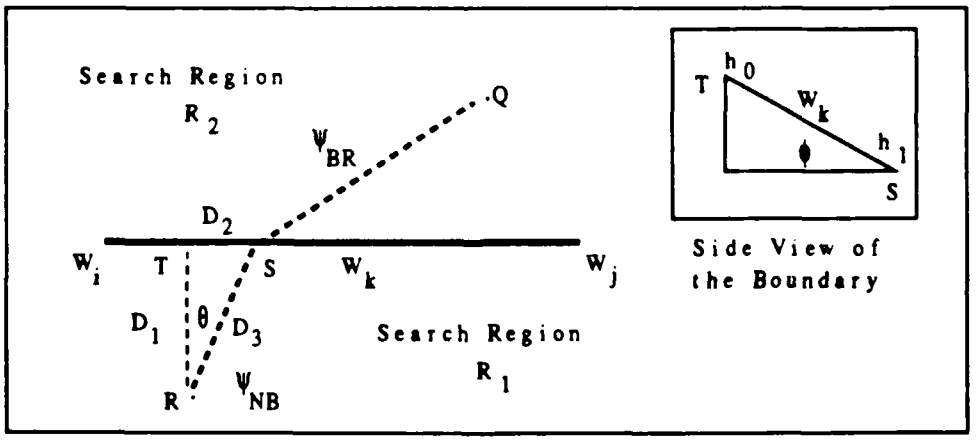

Figure 4.10 Path Behavior for Braking Episodes

The cost formula for optimal path OP is expressed quantitatively as

$$U_{OP} = mg\,\varepsilon D_3 + mg\,\Delta h, \tag{4.18}$$

where $\Delta h = h_1 - h_2$. By geometry, the distance D_3 can be written as

$$D_3 = \frac{D_1}{\cos\theta} = D_1 sec\,\theta, \tag{4.19}$$

and the distance D_2 can be expressed as

$$D_2 = D_1 \tan\theta. \tag{4.20}$$

Letting $h_1 = D_2 \tan\phi$, Eq. (4.18) can be rewritten as

$$U_{OP} = mg\,\varepsilon D_1 sec\,\theta + mgh_0 + mgD_1 \tan\theta\tan\phi - mgh_2. \tag{4.21}$$

To find the minimum cost for the braking and non-braking episodes, it is necessary to differentiate the cost formula with respect to the angle θ and set the resultant expression equal to zero, expressed as

$$\frac{dU_{OP}}{d\theta} = \frac{d}{d\theta}(mg\,\varepsilon D_1(\cos\theta)^{-1} + mgh_0 + mgD_1\frac{\sin\theta}{\cos\theta}\tan\phi - mgh_2) = 0. \tag{4.22}$$

Differentiating, Eq. (4.22) becomes

$$mg\,\varepsilon D_1 \sin\theta\left[\frac{1}{\cos^2\theta}\right] = -mgD_1\tan\phi\left[\frac{1}{\cos^2\theta}\right]. \tag{4.23}$$

Rearranging terms and simplifying, and recalling that $\varepsilon = \tan\theta_{CC}$, Eq. (4.23) becomes

$$\sin\theta = \left[\frac{-\tan\phi}{\tan\theta_{CC}}\right]. \tag{4.24}$$

Thus, the single value, non-braking heading can be written as

$$HUB_{i-1,j} = \theta = \sin^{-1}\left[\frac{-\tan\phi}{\tan\theta_{CC}}\right]. \tag{4.25}$$

The maximum value for θ occurs when the slope of the search window is equal to the critical coasting angle, i.e., $\theta_{CC} = \phi$.

QED.

The single-heading range produced in the type-I region for RBC_{I-IV} is analogous to the critical stability (degenerate) range produced in the type-II region for RBC_{I-II}. Hence:

$$HC_{(1)23} = HP_{23}. \tag{4.26a}$$

$$HC_{12(3)} = HP_{12} \cap HUB_{12}. \tag{4.26b}$$

4. II-II RB Constraints

Figure 4.11 illustrates a type II-II region-boundary constraint RBC_{II-II}. The behavior of an optimal path for RBC_{II-II} is described by the following theorem.

Theorem 4.6: If optimal path OP executes a type-II traversal in region R_{k+1} and a type-II traversal in region R_k, then the heading associated with the type-II traversal in region R_k must be impermissible in region R_{k+1} and conversely, the heading associated with the type-II traversal in region R_{k+1} must be impermissible in region R_k.

Proof: The proof is trivial. Consider an infinitesimal shift in the type-II heading in region R_k to a permissible heading in that same region. Therefore, the traversal in region R_k becomes type-I and the proof is the same as Theorem 4.5. A similar argument can be made for the type-II heading in region R_{k+1}. **QED.**

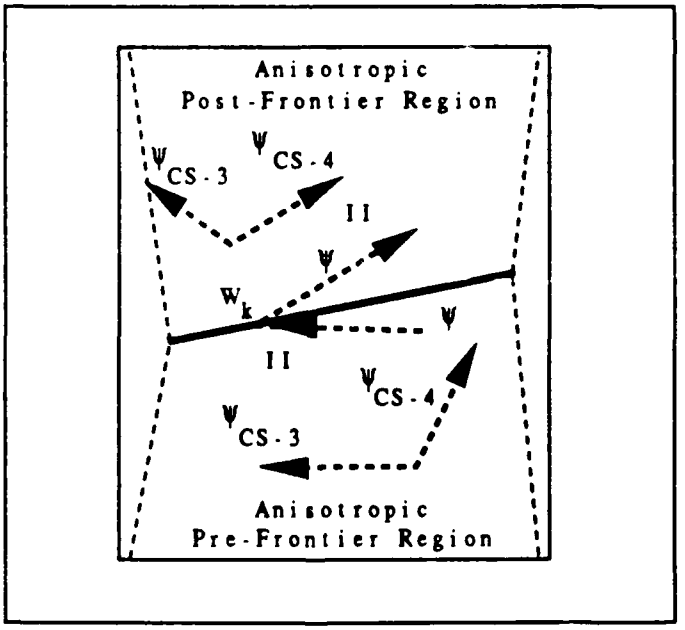

Figure 4.11 Type II-II Region-boundary Constraint

Thus, the constraints are:

$$HC_{(1)23} = HP_{23} \cap HI_{12} \tag{4.27a}$$

and

$$HC_{12(3)} = HP_{12} \cap HI_{23}. \tag{4.27b}$$

5. II-IV RB Constraints

The type II-IV region-boundary constraint RBC_{II-IV} is included only for theoretical completeness. In reality, the constraint can be largely ignored because it occurs in an infinitesimal fraction of natural terrain. The application of RBC_{II-IV} requires two real numbers (the type-II critical path heading and the Theorem 4.5 heading) be equal when picked at random.

6. IV-IV RB Constraints

The final RB constraint addresses consecutive braking traversals. Figure 4.12 illustrates a type IV-IV region-boundary constraint RBC_{IV-IV}. There are two cases to consider: intersecting braking heading ranges and non-intersecting braking heading ranges. The following theorem describes the behavior of an optimal path for RBC_{IV-IV} in the intersecting range case.

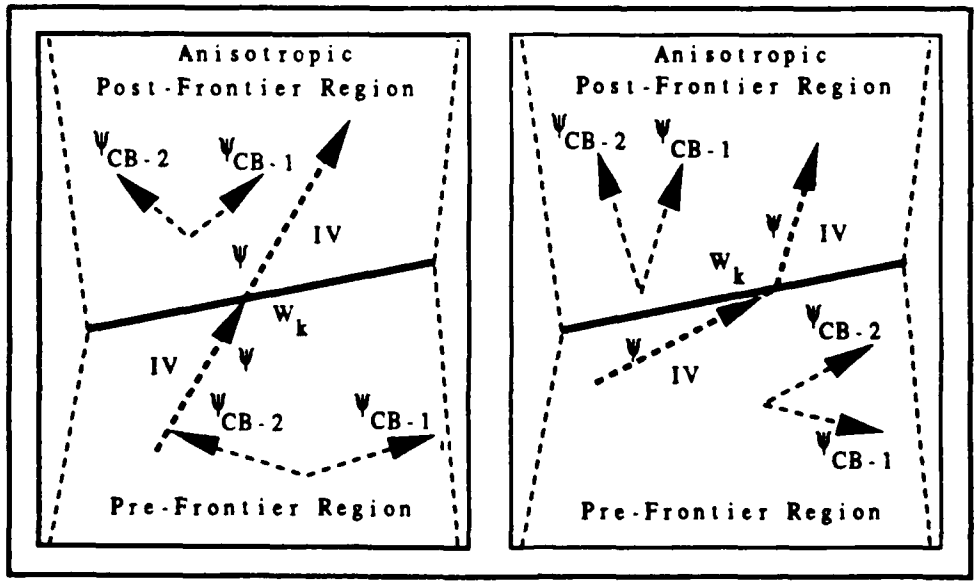

Figure 4.12 Type IV-IV Region-boundary Constraint

Theorem 4.7 If optimal path OP executes consecutive type-IV traversals across regions R_k and R_{k+1} respectively, such that the braking heading ranges in the regions are intersecting, then the path cannot turn on the boundary W_k between the regions.

Proof: Let $PE_1{:}(P_a, P_b)$ represent the straight-line path OP_1 formed by two non-turning consecutive type-IV traversals. Let $PE_2{:}(P_a, P_c)$ and $PE_3{:}(P_b, P_c)$ represent the path OP_2 formed by two consecutive type-IV traversals that turn at the boundary crossing. By Definition 4.2d, type-IV traversals must be braking episodes, and therefore, by Basis Case 1b, Theorem 4.1, OP_1 must have a lower second-order cost than OP_2. Thus, the consecutive type-IV traversals cannot turn on the boundary.

QED.

Thus, if the ranges of braking headings in the regions intersect, the situation is analogous to two type-I traversals. If the braking ranges are non-intersecting, then to minimize second-order costs, both headings are constrained to be critical braking headings. The constraints for both cases are:

$$HC_{(1)23} = HP_{12} \cap HP_{23} = HC_{12(3)} \tag{4.28a}$$

if nonempty, else

$$HC_{(1)23} = HP_{23} \tag{4.28b}$$

and

$$HC_{12(3)} = HP_{12}. \tag{4.28c}$$

HP_{23} is the single-heading (degenerate) braking range or critical braking heading for the region bounded by windows 2 and 3 nearest a braking heading for the region bounded by windows 1 and 2. HP_{12} is the single-heading (degenerate) braking range or critical braking heading for the region bounded by windows 1 and 2 nearest a braking heading for the region bounded by windows 2 and 3.

7. RB Constraints for Vertex Windows

The RB constraints for vertex window expansions are different for RBC_{I-I}, RBC_{I-IV} and RBC_{IV-IV}. For RBC_{I-I}, the restriction on turning at boundaries is eliminated. The optimal path can follow any non-braking heading (type-I traversal) from the vertex window. The same situation exists for RBC_{IV-IV}. This constraint is similar to RBC_{IV-IV} for non-intersecting heading ranges for the edge window expansion. The restrictive constraint of RBC_{I-IV} is also relaxed for the vertex window expansion, that is, the application of the constraint does not generate a single type-I heading as in the edge window counterpart.

For a vertex window expansion, there are three permissible heading ranges for the post-frontier region: (1) non-braking, (2) braking, and (3) stability-constrained. The constraints can be expressed as:

$$HC_{(1)23} = HP_{23} \tag{4.29a}$$

and

$$HC_{12(3)} = HP_{12}. \tag{4.29b}$$

E. CONTROL STRATEGY

Finding the optimal (minimum-energy) path between two points on the two-dimensional map necessitates a search over the set of paths between the two points. This set can be partitioned into "well-behaved" subsets using the polygonal tiling of the map.

Definition 4.3: Any sequence of search windows beginning with the start point and terminating with the goal point such that each pair of windows in the sequence has mutual visibility, is a *goal-feasible window list FWL.*

Given two vertex windows within a goal-feasible window list, the set of all paths that begin at the first vertex window, terminate at the second vertex window, and pass through an identical sequence of polygonal edge windows with the same sequence of traversal types is defined as a *well-behaved subspace of paths WSP.* The convexity theorem, given in [Ref. 3], states that for "well-behaved" subspaces of the

Table 4.2 REGION-BOUNDARY CONSTRAINTS

	Type-I (Exit)	**Type-II (Exit)**	**Type-IV (Exit)**
Type-I (Entry)	$\psi_{ENTRY\ IS\ NON-BRAKING}$ $\psi_{EXIT\ IS\ NON-BRAKING}$ $\psi_{ENTRY} = \psi_{EXIT}$	$\psi_{ENTRY\ IS\ NON-BRAKING}$ $\psi_{EXIT\ IS\ NON-BRAKING}$ ψ_{ENTRY} is impermissible in exit region	$\psi_{ENTRY\ IS\ NON-BRAKING}$ $\psi_{EXIT\ IS\ BRAKING}$ $\psi_{ENTRY} = HUB$
Type-II (Entry)	$\psi_{ENTRY\ IS\ NON-BRAKING}$ $\psi_{EXIT\ IS\ NON-BRAKING}$ ψ_{EXIT} impermissible in entry region	$\psi_{ENTRY\ IS\ NON-BRAKING}$ $\psi_{EXIT\ IS\ NON-BRAKING}$ ψ_{ENTRY} impermissible in exit region ψ_{EXIT} impermissible in entry region	$\psi_{ENTRY\ IS\ NON-BRAKING}$ $\psi_{EXIT\ IS\ BRAKING}$ $\psi_{ENTRY} = HUB$ HUB impermissible in entry region
Type-IV (Entry)	$\psi_{ENTRY\ IS\ BRAKING}$ $\psi_{EXIT\ IS\ NON-BRAKING}$ $\psi_{EXIT} = HUB$	$\psi_{ENTRY\ IS\ BRAKING}$ $\psi_{EXIT\ IS\ NON-BRAKING}$ $\psi_{EXIT} = HUB$ impermissible HUB impermissible in exit region	$\psi_{ENTRY\ IS\ BRAKING}$ $\psi_{EXIT\ IS\ BRAKING}$ $\psi_{ENTRY} = \psi_{EXIT}$ if intersecting HP
	ψ = heading (azimuth)	$HUB = \sin^{-1}(-\tan\phi / \tan\theta_{CC})$	

space of all paths on a two-dimensional, topographic map plane consisting of isotropic and anisotropic polygonal regions with a uniform coefficient of motion resistance, the total path cost is a convex function of parameters sufficient to uniquely specify the path. Convexity of total path cost implies that there is at most one path within the well-behaved subspace of paths that is a local minimum with respect to total cost and therefore, must be the global minimum. The optimization can be performed by bisection iteration. Figure 4.13 illustrates the concept of a goal-feasible window list and well-behaved subspaces of paths. For specific start and goal locations, there is a finite set of goal-feasible window lists. Each goal-feasible window list determines a well-behaved path subspace containing at most one locally optimal path. The least-cost path within the set of locally optimal paths is the globally optimal path.

For a given search window, the process of generating the next visible window is defined as a *search window expansion*. Obtaining goal-feasible window sequences requires search-window "expansions" or repetitive application of the appropriate *successor functions*. A distinct successor function is defined for each region-boundary constraint and is instrumental in the transition process from one search window to the next. Let C_i denote the region classification of search region R_i. Let H_i and T_i represent the permissible heading range and traversal type, respectively, of an optimal path traversal across region R_i.

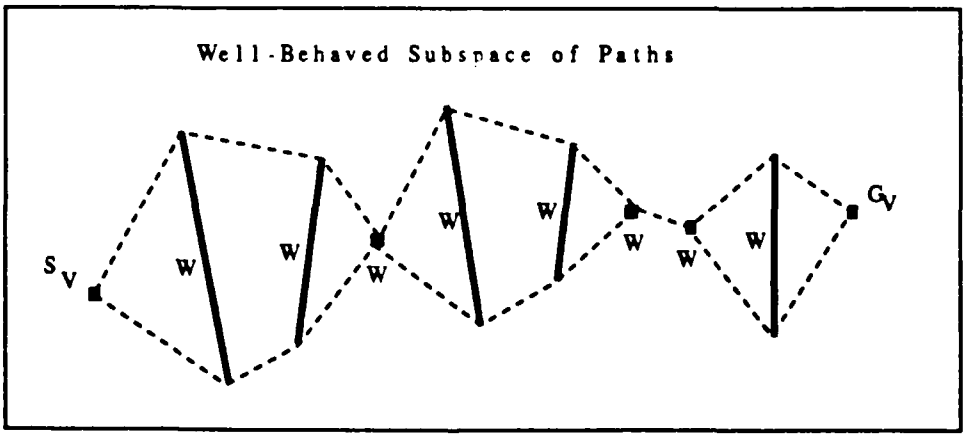

Figure 4.13 Goal-feasible Window List

Let U_i stand for the lower-bound cost estimate. Then, a state description (search node) for a k-window, well-behaved path subspace is specified as

$$\{(W_1,\ldots,W_k,W_{k+1}),(R_1,\ldots,R_k),(C_1,\ldots,C_k),(H_1,\ldots,H_k),(T_1,\ldots,T_k),(U_1,\ldots,U_k)\}. \quad (4.30)$$

W_k represents the current window, W_{k+1} is the post-frontier search window, W_{k-1} is the pre-frontier search window.

Search nodes are maintained on an *agenda*. New search nodes are created from old nodes by *expansion*, adding new items to the ends of each sublist in the state description. The top-level control of the search node expansion proceeds as follows: (1) find a post-frontier window visible from the frontier window, (2) find a possible traversal type for the post-frontier window, (3) determine heading ranges and cost bounds to the post-frontier window, and (4) add the post-frontier window to window sequence. The number of new search nodes generated during an expansion depends on the "branching factor". This is a function of the total number of visible windows in the post-frontier region. An expansion resulting in the empty set terminates the path at that point. Figure 4.14 provides a description of the entire algorithm to find optimal paths.

1. Initialization

The optimal-path-planning algorithm requires certain information to search the state space. Map data, vehicle data, and mission data define the "planning concept" for the minimum-energy path-planning problem. First, a terrain map and vehicle type are selected by the user. The information on vehicle type and surface composition of the terrain establishes the optimal cost rate (coefficient of motion resistance) and region classifications for the problem. Next, the start and goal locations on the map are selected and elevation and visibility information computed. The initial agenda consists of a single search node. From Eq. (4.30), it can be expressed as

$$\{(S_V),(\,),(\,),(\,),(\,)\}. \quad (4.31)$$

After an expansion of the starting state, a typical search node may appear as

$$\{(S_V, W_6),(IS),(I),(((315, op),(45, op)))\}, \quad (4.32)$$

where IS stands for an isotropic region class. An expansion of the above state may generate a search node such as

$$\{(S_V, W_6, W_{10}),(IS, AP),(I, II),(((315, op),(45, op)),((20, cl)))\}, \quad (4.33)$$

where *AP* represents an anisotropic-partially-safe region class. The above search node describes a well-behaved subspace of path traversals from S_V to W_6 and then to W_{10}.

2. Generation of Feasible Window Lists

Goal-feasible window list generation requires expansions of search nodes on the agenda until a valid sequence of windows from start to goal is obtained. The strategy is a modified A^* search over sequences of windows. Similar approaches can be found in Richbourg [Ref. 2] and Rowe [Ref. 4]. In A^* search, the successor function always expands the most promising node on the agenda first. For this, sorting the agenda helps.

```
ALGORITHM Anisotropic-polygonal Path Planning
    Initialize
    Loop
    { until agenda is empty }
        Expand Best Search Node on the Agenda
        For
        { each successor search node generated }
            If
            { search node represents a goal-feasible window list (i.e., goal point is last window) }
                If
                { current optimal path cost > lower bound goal-feasible window list cost }
                    Decompose Goal-Feasible Window List Into Analyzable Pieces
                    For
                    { each subproblem }
                        Generate Locally-Optimal Path Segment
                    End
                    Synthesize Locally-Optimal Path Segments
                    If
                    { new total locally-optimal path cost < current optimal path cost }
                        Record New Path and Cost
                    End
                End
            Else If
            { current optimal path cost > lower bound goal-feasible window list }
                Add Search Node to Agenda
            End
        End
    End
    Return Globally-Optimal Path and Cost
End
```

Figure 4.14 Optimal-path-planning Algorithm

The A^* search strategy requires the sum of a cost function and a heuristic evaluation function. The agenda is sorted by these sums. The cost function used here is a lower bound on the cost from the start point to the current frontier search window. The evaluation function used is a lower-bound estimate of the cost from that window to the goal. Since windows can be edges, the distances (and therefore, costs) between windows lie within bounds. Given a vertex window W_i and an edge window W_m with endpoint-vertex windows W_j and W_k, the lower-bound distance between W_i and W_m is the minimum of three values: (1) the distance between W_i and W_j, (2) the distance between W_i and W_k, or (3) the distance between W_i and the intersection of the perpendicular projection of W_i onto W_m, if one exists. The lower-bound distance between two edge windows is determined in a similar manner. It is the minimum of eight distances: the four distances obtained by connecting all combinations of the endpoint-vertex windows between the two edge windows and the four distances obtained from the perpendicular projections of each endpoint-vertex window on the opposite edge window as above. Once a lower-bound distance is determined, a lower-bound cost can be computed using the optimal cost rate (minimum value for the coefficient of motion resistance). For type-IV (braking) traversals, a different lower bound cost function is used; that is, the minimum elevation difference between the two windows. For two edge windows, the cost is computed as the minimum of four values; the elevation differences obtained from all combinations of the endpoint-vertex windows between the two edge windows. For edge window and vertex window combinations, it is the minimum of two elevation differences. The straight-line Euclidean (lower-bound) distance to the goal was used for the evaluation function. Traversal types are not considered in the evaluation function because various types of regions can be crossed enroute to the goal. The sum of the minimum costs between each pair of successive windows in the window list was used for the cost function.

After picking the best agenda node, the successor functions generate every possible successor node of it from the visibility, traversal type, and heading information available using the constraints discussed in Section IV.B. For instance, a type-I traversal in a pre-frontier region can give up to four distinct successor nodes; one *type I-I*, two *type I-II* (one for each symmetric critical stability heading), and one *type I-IV*. As the A^* search continues, there is opportunity to prune nodes based upon the visibility and heading restrictions discussed earlier. The search eventually generates a sequence of windows that terminates with the goal, i.e., a goal-feasible window list. The first goal-feasible window list obtained represents the sequence of search windows from the start to the goal having the best lower-bound cost but not necessarily the best true cost. Since the algorithm uses "bounded" costs instead of exact costs, the search cannot necessarily terminate after finding the cost of the optimal path within the first goal-feasible

window list. This cost must be compared to all lower-bound costs (cost function and evaluation function) of items remaining on the agenda. Thus, the cost of the optimal path within the goal-feasible window list is an upper bound on the globally-optimal-path cost. This cost can be used to guide the search and prune nodes on the agenda when necessary. Any search node on the agenda with a lower-bound cost exceeding this upper bound cost can be eliminated from the agenda. Analogously, a goal-feasible window list with a lower-bound cost exceeding the current upper-bound cost can be pruned immediately. The search terminates when the lower-bound cost of every item on the agenda exceeds the "best" globally-optimal-path cost (upper bound) found thus far, or the agenda is exhausted.

3. Decomposition of Feasible Window Lists

Within a goal-feasible window list, two consecutive vertex windows are a *deterministic path segment*, or a *solved path segment*. The segment is considered "solved" because the optimal path must travel at a single, unique heading between the vertex windows. Other pairs of windows are termed "unsolved" and the optimal path through them must be found by iterative optimization. The problem of finding the optimal path within a goal-feasible window list is simplified by: (1) extracting the deterministic path segments (solved subproblems), and (2) eliminating unnecessary edge-window boundaries that do not affect the search process. This accomplished, the complete optimal path for the goal-feasible window list can be determined by connecting the optimal paths found by independently computing the locally-optimal paths for the set of connected, unsolved subparts of the window sequence. Figure 4.15 shows a goal-feasible window list decomposed into its constituent subproblems.

There are two ways to eliminate edge windows. A portion of a goal-feasible window list with consecutive type-I traversals across each region can be treated as a single search region with a region-boundary-constraint heading range computed from Eqs. (4.16a) and (4.16b). Since RBC_{I-I} does not allow an optimal path to turn on a boundary crossing, the intermediate edges in the window sequence can be eliminated without consequence. An optimal path needs only make one type-I traversal across the newly-formed region. A similar consolidation can occur with consecutive type-IV traversals with intersecting region-boundary-constraint heading ranges. The justification for the type-IV consolidation is based on the premise that an optimal path executing consecutive braking traversals with overlapping heading ranges never turns on a boundary crossing. The traversal cost is computed simply as the difference in elevations from the start of the first braking episode to the end of the last braking episode. Therefore, intermediate edges can be eliminated without affecting the iterative optimization. There is no analog for a type-II consolidation.

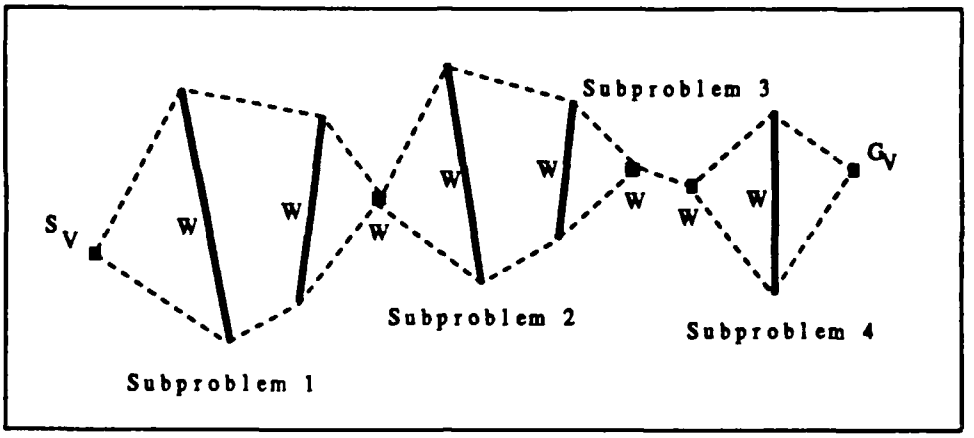

Figure 4.15 Feasible Window List Subproblems

4. Generation of Optimal Paths

There are two steps in generating an optimal path within an unsolved path segment. The process of generating a search node containing a goal-feasible window list produces a list of permissible heading ranges in the regions. A permissible heading range, as it exits a frontier search window, casts a "shadow" on the window. This is the section of the original window through which an optimal path can pass. The set of all shadows, together with the entry and exit vertex windows, is an *optimization corridor*. Figure 4.16 illustrates an optimization corridor.

To find the turning points on each window within the corridor to minimize total path cost, iterative optimization is employed. This technique is a form of minimax search or bisection iteration and can be employed because the path cost is a convex function [Ref. 3]. It is also referred to in [Ref. 5] as "interval halving". Once the goal-feasible window list has bracketed the optimal path, the interval reduction method continuously refines the estimate of the true optimal path within the corridor varying only the position of the turn point along the edge window. The initial estimate of the turn points at each boundary crossing within the corridor is assumed to be the midpoint of the edge window unless a stability constraint (type-II traversal) dictates otherwise. At each iteration, exactly one-half of the search window is eliminated. Since the midpoint of subsequent intervals on the edge window is equal to one of the previously-computed trial points, only two evaluations are required at each iterative step.

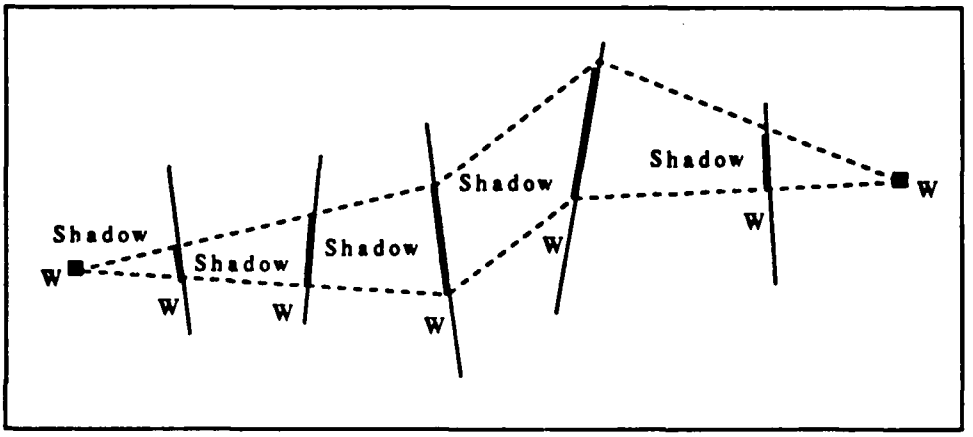

Figure 4.16 Optimization Corridor

F. SUMMARY

The optimal-path-planning algorithm provides a way to find minimum-energy paths in natural terrain. It uses the mathematical model of vehicle-terrain interaction from Chapter III and consists of a comprehensive problem representation, a description of optimal path behavior, and a search strategy. The problem representation is divided into two components: a state description and a set of successor functions to provide movement between states. A key element in the state description is the "search window" or the entity through which an optimal path must pass. The geometric structure of the two-dimensional map plane facilitates visibility analysis between windows. This limits potential successor states in the search space. Permissible heading ranges for optimal paths using stability and braking constraints are stored. The headings provide a second method to reduce the search space by limiting the range of path traversals along search windows.

After the search space is pruned through visibility and heading analysis, the behavior of optimal path segments between windows is described by a set of path traversal types. A set of successor functions for a partial window list gives "expansions" of that list based on traversal-sequencing constraints. The control strategy is a modified A^* search that finds lower-bound sequences of windows from start to goal, i.e., goal-feasible window lists. The lists are reduced to simpler subproblems about "deterministic path segments" that are solved independently by iterative optimization and then linked. The cost of the path found gives a global upper bound on the optimal path and guides the remainder of the search. The A^*

search continues to generate possible sequences of windows from start to goal until the agenda is exhausted.

V. DEMONSTRATION

A. INTRODUCTION

The mathematical model of vehicle-terrain interaction and the optimal-path-planning algorithm have been implemented in a computer program. The program uses a "synthetic terrain map" consisting of a set of symmetric polygonal regions. These regions represent the discretization of a natural terrain surface into a polyhedral structure and the subsequent projection into the two-dimensional topographic map plane as discussed in Chapter III.

The LISP programmimg language [Ref. 6] was selected for the implementation due to its flexibility in data structures and rapid prototyping capability. The dialect chosen was *Common LISP* because of its emergence as the "standard" LISP programming language and also for portability considerations [Ref. 6]. The functional nature of the language facilitated the hierarchical development of a large program. Individual components of the program were constructed and tested independently.

The programming environment is a Symbolics 3675 computer employing version 7.1 of the Genera operating system. The Symbolics system was chosen because of its integrated developmental environment and processing speed. The high-resolution monitor and built-in graphics packages enabled the user interface to be constructed with minimal difficulty.

B. IMPLEMENTATION

1. Constructing the Terrain Map

The terrain map was constructed in a hierarchic manner using the built-in LISP construct called *structure*. The structure is a "generic" object with a set of associated attributes that describe the object and is similar to a database schema or a LISP "property list". The LISP structures have built-in access functions to retrieve attribute values rapidly. Instances of the defined structures are created as needed by the program. Three structures are created for the synthetic map: (1) a vertex structure, (2) an edge structure, and (3) a region structure. Figure 5.1 provides a specification for the structures. Using a data compression technique, the vertices are described by the x, y, and z coordinates [Ref. 7]. Edges are defined in terms of the endpoint vertices and the regions are constructed in terms of the boundary edges. This method avoids the data duplication by defining each particular vertex only once. An edge structure also records information about adjacent regions. The region structures contain attribute information on slope,

VERTEX:
(x-coord y-coord z-coord edge-list visibility-list)
EDGE:
(vertex-list adjacency-list visibility-list)
REGION:
(edge-list slope orientation surface-composition type stability-constraints braking-constraints)

Figure 5.1 LISP Structures for Map Representation

orientation, surface composition, surface covering, type of region, and (vehicle-specific) braking and stability constraints. Once the set of polygonal regions has been constructed, the concave background region is partitioned into a set of convex polygonal regions each of which is isotropic. The partitioning facilitates visibility analysis. There are standard algorithms available for this type of decomposition [Ref. 8].

Currently, the map is created by an interactive input routine. However, work has been work undertaken to construct the polygonal regions from a gridded set of data points automatically [Ref. 9]. This is a difficult task since the set of grid points must be fitted to a plane and boundaries constructed to create a geometrically consistent mesh. The description of the mesh in terms of its constituent vertices, edges, and regions defines the map input necessary for creating the search space, i.e., the search windows and search regions.

2. Constructing the Vehicle Concept

The physical properties of the vehicle, relevant to the minimum-energy path-planning problem, are defined in the *vehicle concept*. A LISP structure is used to record these properties as illustrated in Figure 5.2. The properties of the vehicle are used to produce selected values for the region attributes listed above; that is, the coasting and gradient slopes define the isotropic regions and obstacle regions respectively, as discussed in Chapter III. The stability safety margin is used to estimate the critical stability angles. Once a vehicle concept is selected, a pre-processing routine creates a *context-dependent map*. This

VEHICLE:
(name type weight center-of-gravity coasting-slope contour-slope gradient-slope stability-safety-margin)

Figure 5.2 LISP Structure for the Vehicle Concept

map contains a set of homogeneous mobility regions that retains the terrain-dependent and vehicle-terrain dependent information on the respective structures.

Three military vehicles served as prototype agents for the program: M113-APC Armored Personnel Carrier (tracked), M-966 Armored Tow Carrier (wheeled), and M-813 Cargo Truck (wheeled). Actual data on vehicle weight, gradient slope, and contour slope was obtained from Department of the Army Technical Manuals [Ref. 10-12]. Data on critical coasting slopes was not available since it must be obtained empirically through a series of controlled tests on the actual vehicles. Therefore, several different values were used as reasonable estimates for the purpose of the program. The estimates were based on personal experience in operating wheeled and tracked vehicles in an off-road environment.

3. Spatial Reasoning Functions

To manipulate the map structures and identify certain key spatial relationships such as "connectivity" and "containment" requires a set of *spatial reasoning* functions. The functions perform edge and vertex visibility operations, location-finding, distance calculations, and heading analysis. The convex nature of the polygons that tessellate the map simplify the spatial reasoning functions especially for visibility operations. Spatial reasoning functions are partitioned into two groups: (1) functions that return a specific map value, e.g., *get-region-from-point*, *get-distance-to-edge*, etc., and (2) functions that test a particular condition (predicates), e.g., *anisotropic-safe-region-p*, *obstacle-edge-p*, etc.. These functions are the *primitive* operations that can be applied to the terrain map representation.

4. Search Functions

The search functions are divided into three separate groups. The first group is responsible for conducting all vehicle heading analysis operations including geometric, stability, braking, instability, and non-braking criteria. The second group focuses on building the initial agenda and creating new search nodes through the set of successor functions discussed in Chapter IV. The final set of search routines provides higher-level operations such as goal-feasible window-list generation, problem decomposition, iterative optimization, and path synthesis.

5. Command-and-Control Functions

The command-and-control module is the nucleus of the path-planning program. There are three fundamental tasks that must occur. The first task involves selection of a terrain map. Once selected, the map is loaded from disk and the structures generated to build the terrain. Next, a vehicle concept is selected and vehicle-terrain dependent map information is updated on the appropriate region structures. Finally, a set of functions assert a vehicle mission by defining start and goal points on the topographic map.

The start and goal can be chosen at any desired map location except within obstacle regions. A query must be made to determine which search regions contain the start and goal. Elevations within the regions are also computed. Visibility from the start and goal to adjacent windows is obtained and all structures are updated as needed. If all relevant information is provided, search for the minimum-energy path begins. Incremental paths (locally optimal paths within goal-feasible window lists) are displayed as computed along with other statistical information (Section V.D).

C. TEST DATABASES

For the prototype implementation, the terrain database consists of the hierarchically-constructed synthetic map stored as instances of LISP structures. The LISP structures, after initial creation, are saved on disk and can be regenerated each time the terrain map is used. The tested terrain is a set of multi-level, truncated pyramids. The elevation of each vertex (z-coordinate) can be controlled to vary the slopes and orientations of the polygonal faces of the polyhedron. This does not affect the shape of the polygon face in the two-dimensional projection, but rather influences the relative slope of the region. Thus, by changing the z coordinates and recomputing the slopes and orientations, the geometric configuration can model truncated (pyramidal) hills, valleys, ridgelines, ravines, or saddles. The multi-level (terraced) effect is employed to maximize the number of different types of regions that are crossed to verify the effects of all possible combinations of region-boundary constraints.

Two databases that provide digital terrain information at a grid resolution of 12.5 and 100 meters have been obtained from the Defense Mapping Agency. Efforts to generate the appropriate polygonal format are underway at the Naval Postgraduate School (NPS) and the results will used in the next generation of the computer program. This work, however, is not part of this dissertation.

D. RESULTS

The minimum-energy path-planning problem is a member of a family of path-planning problems denoted as the "combinatorial shortest-path problems" [Ref. 13]. In general, the order of complexity of these problems is exponential in the worst case as a function of the number of the number of vertices in the search space. It is anticipated that the pruning criteria discussed in Chapter IV will allow the algorithm to exhibit better performance in the average case. No formal complexity analysis is attempted in this dissertation.

The results of several runs of the program are shown at Figures 5.3, 5.4, and 5.5. Various routes were selected to include short missions within the same terrain feature and longer missions involving travel between features. The anisotropic nature of the search is graphically illustrated in Figures 5.6 and 5.7 where the start and goal points were reversed, resulting in different minimum-energy paths.

E. SUMMARY

A Common LISP implementation of the theoretical models discussed in the previous chapters has been developed. During the prototype development, emphasis was placed on designing an algorithm that employed the concept of heuristic search over a grid-free terrain representation eliminating unproductive paths through a comprehensive set of pruning criteria. Less attention was given to efficiency issues and remains an extension for future research.

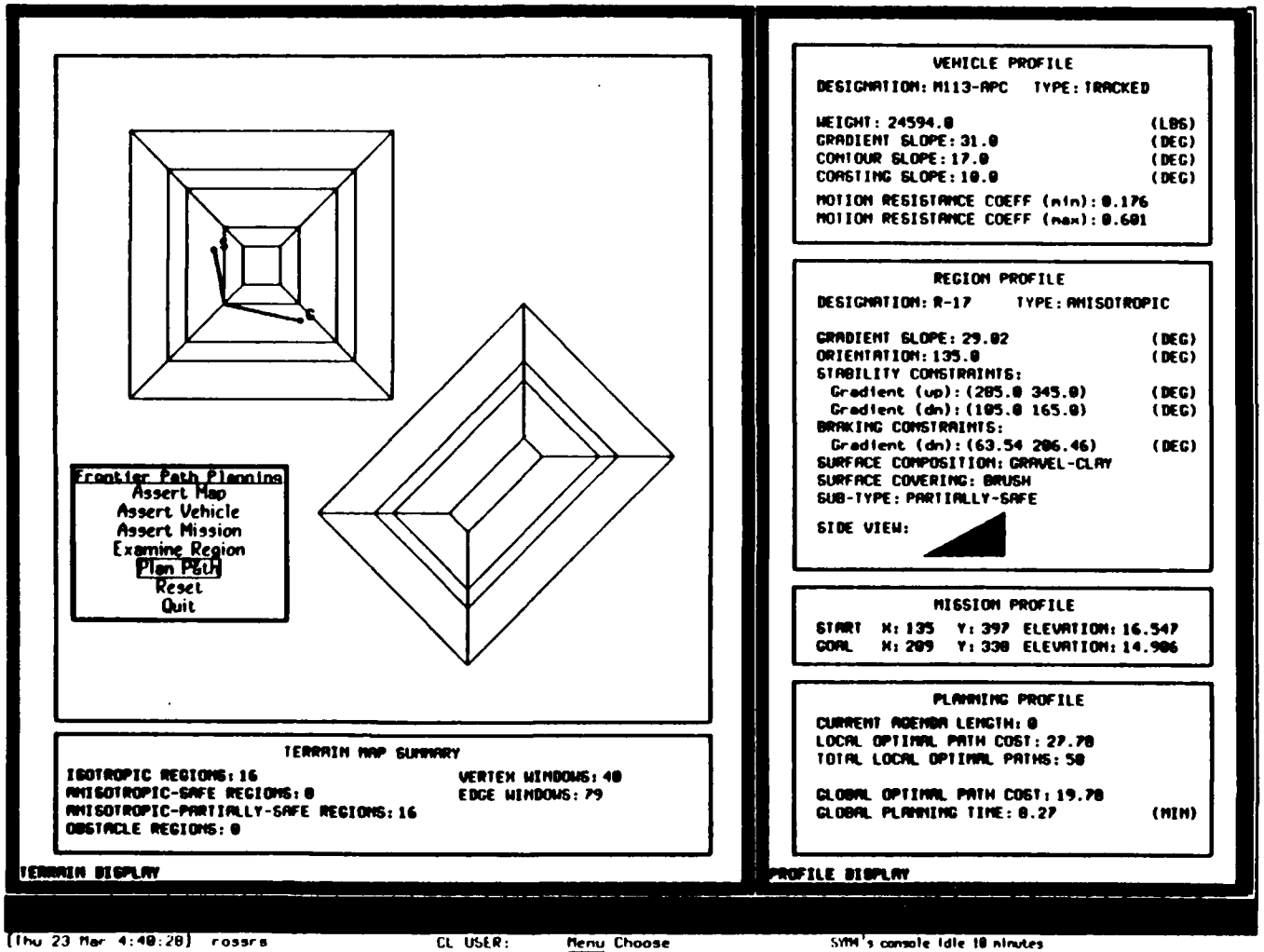

Figure 5.3 Computer Simulation: Path 1

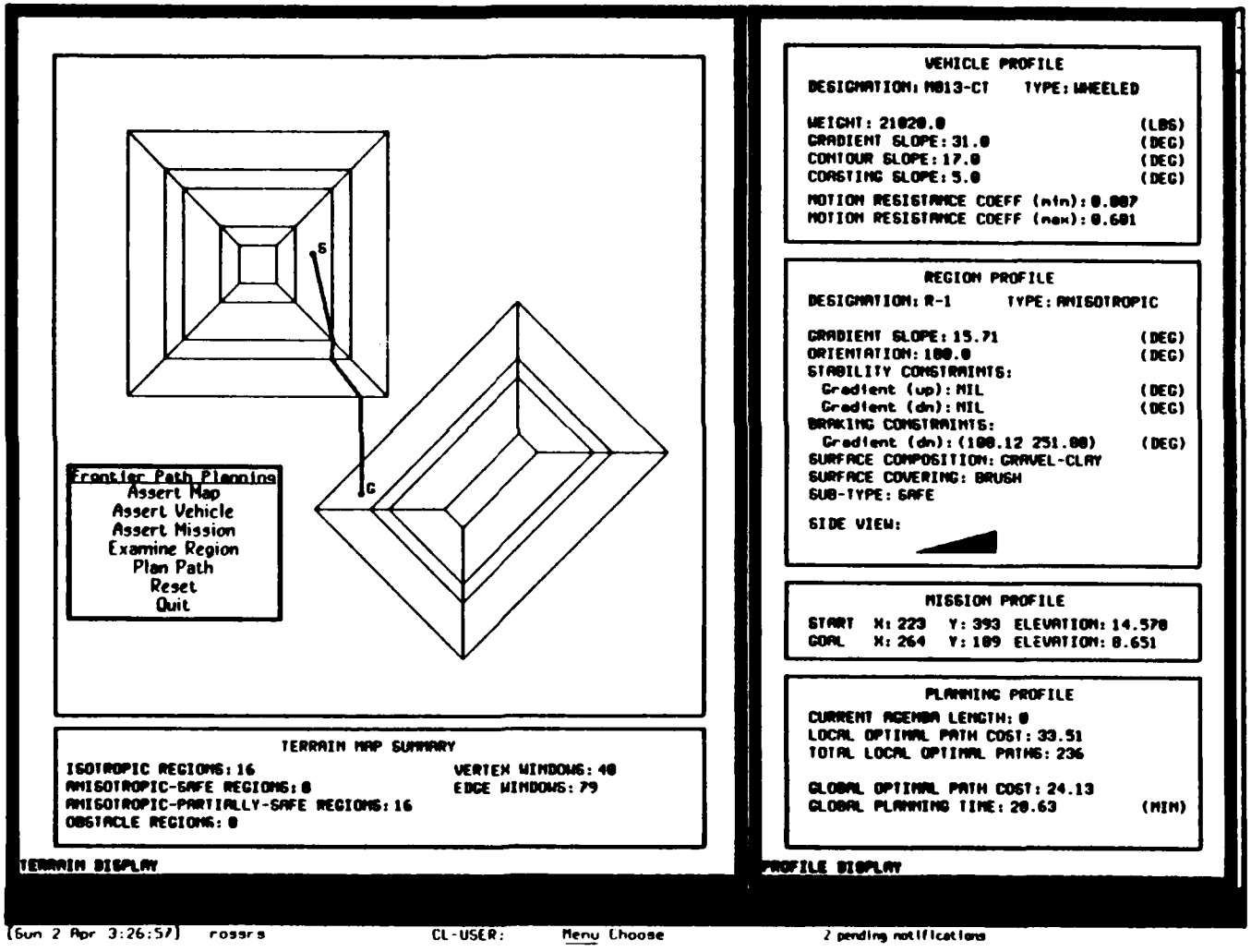

Figure 5.4 Computer Simulation: Path 2

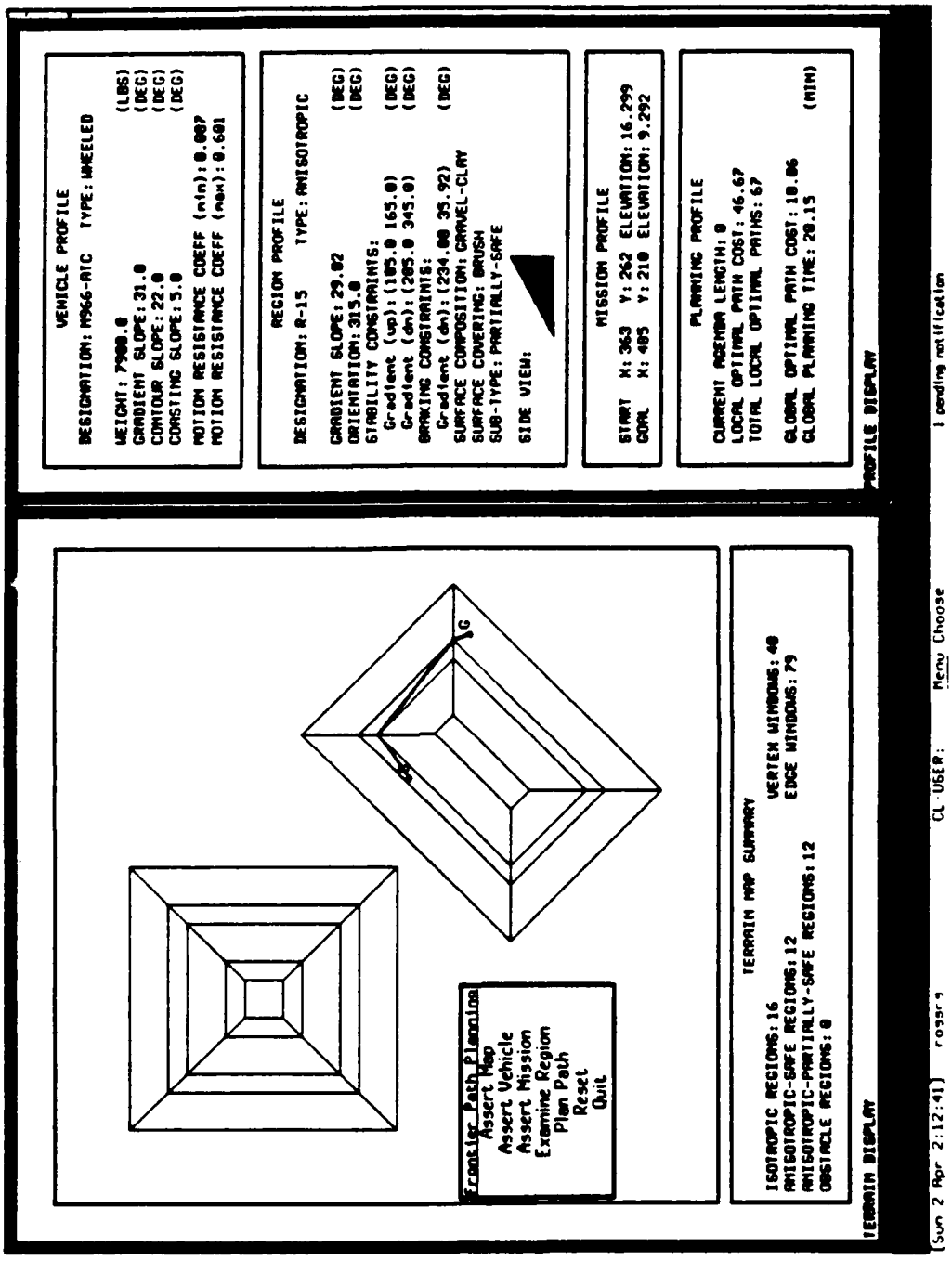

Figure 5.5 Computer Simulation: Path 3

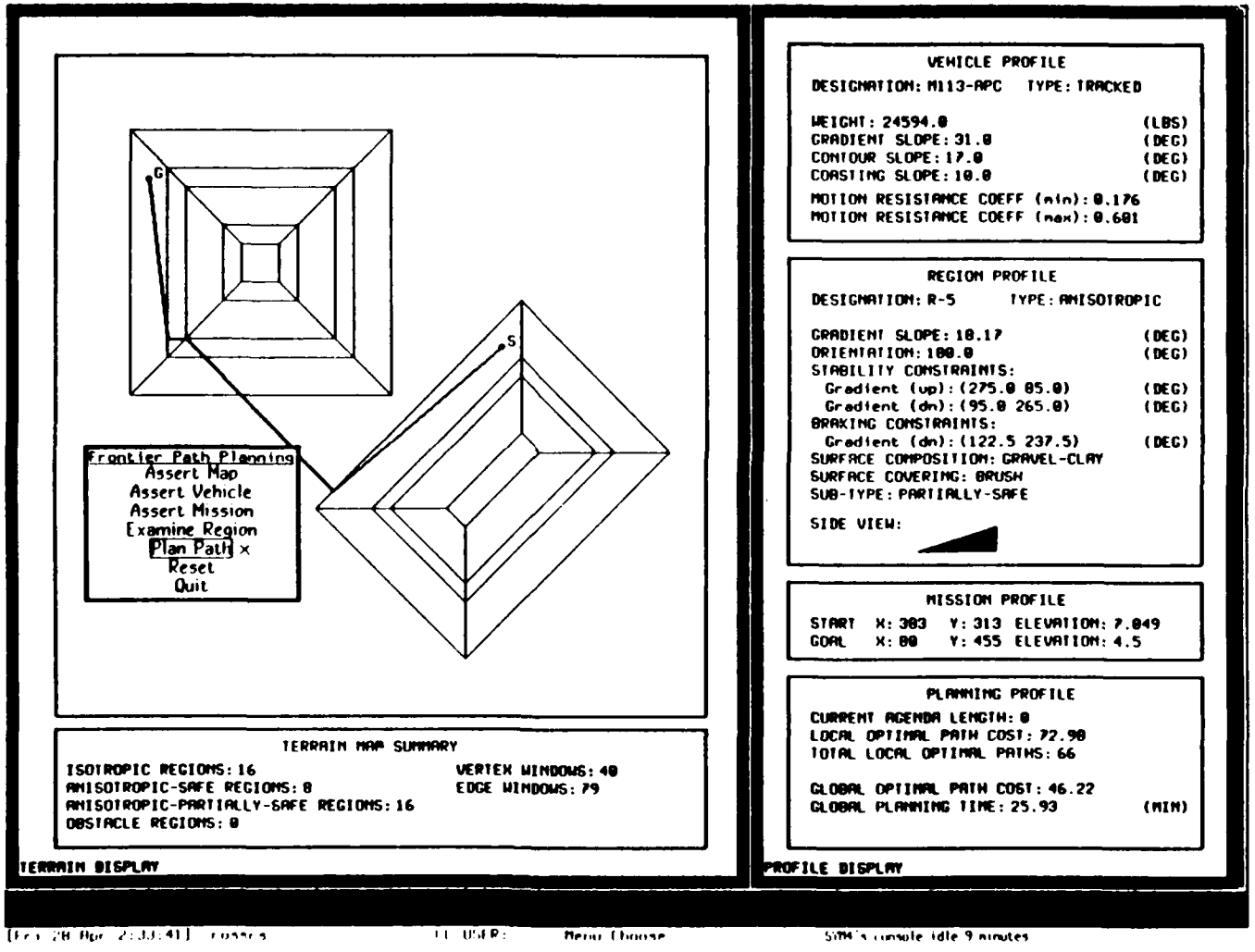

Figure 5.6 Computer Simulation: Path 4

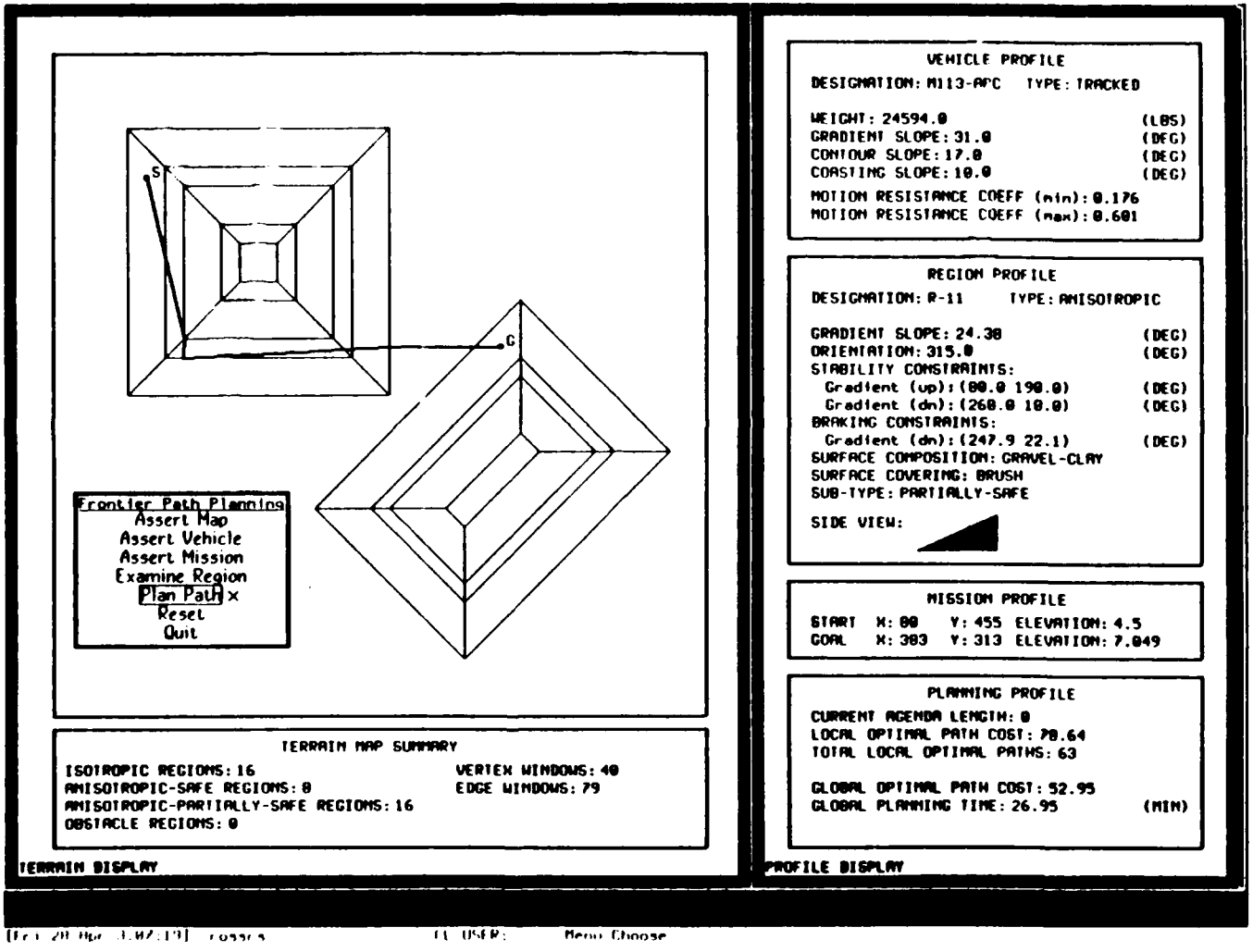

Figure 5.7 Computer Simulation: Path 5

VI. SUMMARY AND CONCLUSIONS

A. RESEARCH CONTRIBUTIONS

In this dissertation, the problem of finding a minimum-energy path across arbitrarily-contoured terrain with direction-dependent traversal costs and motion constraints has been examined. The fundamental contributions of this research are three-fold: (1) the development of a *mathematical model* of vehicle-terrain interaction that predicts the performance of an off-road vehicle when operating in natural terrain, (2) the development of a *symbolic terrain model* that divides the world into meaningful parts suitable for spatial reasoning, and (3) the development and implementation of an *optimal-path-planning algorithm* that exploits the theory of the mathematical model to compute stable, minimum-energy paths.

With few exceptions, previous research in the areas of mobility modelling, terrain representation, and path planning did not focus extensively on developing an integrated approach to the problem of finding optimal routes in an off-road environment. Until now, most of the techniques for path finding have relied on the traditional grid-based approach for terrain representation and for conducting the search. Those that attempted to employ a more symbolic approach in the representation of terrain and apply alternative search strategies such as the ray-tracing approach, posited an isotropic cost function in the generation of the optimal paths, and did not consider the effects of the vehicle stability or braking. Thus, this research represents the first attempt at developing a unified set of models that addresses the problem of finding optimal routes using a combination of symbolic terrain, anisotropic traversal costs, motion constraints, and heuristic search.

The mathematical model posits a simplified view of motion that assumes the force required to move between two points on the terrain surface is due to a "towing force" resulting from the vehicle being pulled by a mythical cable. An analysis of the forces acting on the vehicle results in a set of *energy equations* that separate the resistive energy costs due to Coulomb-friction forces from the potential-energy costs associated with gravity. Since the object is to plan minimum-energy paths over relatively large distances, a low constant speed is assumed for the vehicle and kinetic energy is ignored. Potential-energy costs are factored out as a constant term independent of the global path, and the remaining costs are *resistive* in nature. The resistive-energy costs are a function of the vehicle *motion resistance* represented by a composite resistive coefficient and the straight-line distance in the topographic map plane. The model

considers vehicle braking as a significant parameter and incorporates its effects in the overall estimate of resistive forces. The partitioning of energy costs into resistive-energy costs and potential-energy costs facilitates the development of a two-dimensional path-planning model that incorporates three-dimensional information with respect to motion constraints. The mathematical model also exploits the concept of direction-dependent traversal costs or the principle of anisotropism which is paramount to the computation of minimum-energy paths. The traversal costs can be classified as either braking (heading-dependent) or non-braking (heading-independent).

The symbolic terrain model is a key part of the mathematical model. It allows the continuous nature of the arbitrarily-contoured terrain to be discretized and represented conceptually as a *polyhedron.* Retaining the three-dimensional gradient information for each convex, polygonal face of the polyhedral structure, the components are projected into the two-dimensional plane analogous to the representation of a cartographic map. The resulting polygonal mesh tessellates the map plane into a well-defined set of homogeneous cost regions. The symbolic model classifies each region according to the constraints set forth in the mathematical model, specifically in the areas of braking and stability. The partition divides the terrain into regions of safety and low traversal cost, regions of safety with possible vehicle braking, regions of partial safety with possible braking, and unsafe or obstacle regions. Traversal costs are divided into two conceptually simple categories: costs associated with braking episodes and costs associated with non-braking episodes. The former is a function of the difference in elevations between the start and end of the braking episode and the latter is a function of straight-line distance across the region and the coefficient of motion resistance, uniquely determined for a particular vehicle-terrain combination.

The search space in the minimum-energy path-planning problem centers on the concept of "search windows" representing the physical edges and vertices of the homogeneous mobility regions. This is in contrast to the more traditional uniform-grid approaches that restricted terrain information to sample points at arbitrary resolutions. Reasoning about symbolic objects on the topographic map plane more closely models the way humans think. The symbolic approach is also more computationally accurate in planning optimal paths since the problem of digitization bias is eliminated. The problem of information loss resulting from the imposition of a uniform grid is non-existent.

Given a symbolic terrain representation, new approaches to search-space reduction are proposed using geometric visibility and path heading analysis. The cost criteria and stability constraints from the mathematical model are exploited in the analysis of optimal path behavior. The path-planning algorithm

uses a theory of optimal path behavior applied within the boundaries of the homogeneous search regions and at boundary crossings between regions. The theory posits a small, but mathematically provable number of ways that a path can cross a region based on the aforementioned constraints. Using the well-defined set of path traversal types, a set of region-boundary constraints based on the entry and exit traversal types describes the behavior of an optimal path at window crossings. Permissible heading ranges are computed from the region-boundary constraints. Employing a modified A^* search technique, sequences of windows are generated that begin at the start point and terminate at the goal point. Each of these "goal-feasible" window lists contains, at most, one locally-optimal path. This path is obtained by iterative optimization using a form of minimax (bisection) search and is an upper bound on the globally-optimal path. Continuing the search, every goal-feasible sequence of windows is found and then optimized. Within a window sequence, an optimal path having a lower cost than the upper-bound path cost is retained as the best path. The process repeats until the agenda is empty or the cost of every remaining path is greater than the current upper-bound path cost. The best path of the locally-optimal paths is the globally-optimal path. The algorithm is guaranteed to generate the optimal (minimum-energy) path due to the lower-bound cost and evaluation functions and the exhaustive search of every goal-feasible window sequence.

The body of theory developed in this dissertation has been implemented in a computer program on a Symbolics LISP machine. The implementation produced some interesting results on a synthetic terrain map of symmetric convex polygons. The anisotropic nature of the terrain generated completely different paths when start and goal points were reversed, as predicted by the theoretical model. In general, it was observed that the minimum-energy paths follow longer routes in isotropic regions rather than shorter routes over steeper slopes requiring braking. The degree to which this phenomenon occurs is based on how far "out of the way" the vehicle would have to travel before the non-braking cost would overtake the braking cost.

B. RESEARCH EXTENSIONS

The primary focus of this research has been on developing an integrated approach to solving a minimum-energy path-planning problem. For the most part, efficiency issues were not given a high priority in the development of the search algorithm or in the initial demonstration of the theoretical results. Several areas are worth further exploration. Since line-intersection routines are an important part of the computer program, more efficient algorithms can reduce computation time. Pavlidis [Ref. 14] and Foley and Van Dam [Ref. 7] describe several approaches to implementing efficient line-intersection algorithms.

Efficiency can also be improved by employing parallelism where appropriate and feasible. For example, during the A^* search, after each successful window expansion, a new processor can be assigned to each newly-created search node. Processors assigned to paths that are pruned can be recycled as needed. Processors assigned to paths that result in a goal-feasible window list and locally optimal path can report results to a central location as in the "blackboard model" [Ref. 15].

Other approaches for improving the efficiency of the algorithm involve using different lower-bound cost functions and using previous optimal path information (learning from experience) to assist in computing new optimal paths. Storing previous paths may eliminate needless computation when path-planning operations occur repeatedly within the same map area. Efficiency and speed of computation are critical if the algorithm is to be used for mobile-robot path planning.

Another interesting extension to the current work involves generalizing the approach outlined in this dissertation to include the possibility of multiple soil types and vegetation within the planning space and employing a combination of search techniques within the same problem. This idea has been partially explored by Rowe [Ref. 3].

In the area of terrain modelling, the process of creating a polyhedral terrain model from a uniform, gridded data set can be improved. It is possible that a user-assisted approach may be the most appropriate solution to the problem, in which a computer program creates the initial symbolic terrain map and then employs human intervention to resolve any anomalies.

As discussed in Chapter I, the decision as to which search strategy is "best" depends on many factors. Certain parts of a terrain map may be suited to the wavefront-propagation method using a uniform-grid representation while other sections of the map may be searched more effectively with a polygonal representation using either the "ray tracing" approach or methods discussed in this dissertation. The integration of different path-planning techniques could be controlled by an expert system that would heuristically apply the appropriate search strategy for various types of terrain representations and map complexity. The value and form of this type of "hybrid" search strategy remains an open question.

LIST OF REFERENCES

1. Charniak, E. and McDermott, D., *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company, Reading, MA, 1986.
2. Crowley, J. L., "Path Planning and Obstacle Avoidance," in *Encylopedia of Artificial Intelligence*, ed. S. C. Shapiro, v. 2 , John Wiley and Sons, New York, NY, 1987.
3. Foley, J. D. and VanDam, A., *Fundamentals of Computer Graphics*, Addison-Wesley Publishing Company, Reading, MA, 1984.
4. Gaw, D. and Meystel, A., "Minimum-Time Navigation of an Unmanned Mobile Robot in a 2-1/2D World with Obstacles," *Proceedings of the IEEE Conference on Robotics and Automation*, April 1986.
5. Nilsson, N. J., *Problem-solving Methods in Artificial Intelligence*, McGraw-Hill Book Company, New York, NY, 1971.
6. Richbourg, R. F., *Solving a Class of Spatial Reasoning Problems: Minimal-Cost Path Planning in the Cartesian Plane*, Doctoral Dissertation, Computer Science Department , U.S. Naval Postgraduate School, June 1987.
7. Thorpe, C. E., "Path Relaxation: Path Planning for a Mobile Robot," *Proceedings of the AAAI-84*, 1984.
8. Parodi, A. M., "A Route Planning System for an Autonomous Vehicle," *IEEE Computer Society Conference on Artificial Intelligence Applications*, 1984.
9. Pars, L. A., *An Introduction to the Calculus of Variations*, Heinemann Educational Books, Ltd., London, 1962.
10. Finney, R. and Thomas, G., *Calculus and Analytic Geometry, 5th Edition*, Addison-Wesley Publishing Company, Reading, MA, 1981.
11. Turnage, G. W. and Smith, J. L., *Adaptation and Condensation of the Army Mobility Model for Cross-Country Mobility Mapping*, Technical Report GL-83-12, U. S. Army Engineer Waterways Experiment Station (WES), September 1983.
12. Shelkin, B. D. and Foster, D. D., "Defense Mapping Agency Digital Data Policy," in *Geographic Information Systems in Government*, ed. B. K. Opitz, v. 2 , A. Deepak Publishing, Hampton, VA, 1986.
13. Defense Mapping Agency Hydrographic/Topographic Center, *Product Specification, Digital Land Mass System (DLMS) Data Base*, 1984.
14. Ballard, D. B. and Brown, C., *Computer Vision*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
15. Pavlidis, T., *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, MD, 1982.
16. Yee, D., *Three Algorithms for Planar-Patch Terrain Modeling*, M.S. Thesis, Computer Science Department, U. S. Naval Postgraduate School, June 1988.
17. Zyda, M. J., et al., "Flight Simulators for Under 100,000 Dollars," *IEEE Computer Graphics and Applications*, v. 8, no. 1, January 1988.
18. Kwan, D. T., "Terrain Map Knowledge Representation for Spatial Planning," *IEEE Computer Society Conference on Artificial Intelligence Applications*, 1984.
19. Kwan, D. T., et al., "Natural Decomposition of Free Space for Path Planning," *IEEE Conference on Robotics and Automation*, 1985.
20. Antony, R. and Emmerman, P. J., "Spatial Reasoning and Knowledge Representation," in *Geographic Information Systems in Government*, ed. B. K. Opitz, v. 2 , A. Deepak Publishing, Hampton, VA, 1986.

21. Samet, H., "The Quadtree and Related Hierarchical Data Structures," *Computing Surveys*, v. 16, no. 2, 1984.

22. Minsky, M., "A Framework for Representing Knowledge," in *The Psychology of Computer Vision*, ed. P. Winston , McGraw-Hill Book Company, New York, NY, 1975.

23. Rula, A. A. and Nuttall, C. J., *An Analysis of Ground Mobility Models (ANAMOB)*, Technical Report, M-71-4, U. S. Army Engineer Waterways Experiment Station, July 1971.

24. Knight, S. J. and Rula, A. A., "Measurement and Estimation of the Trafficability of Fine Grained Soils," *Proceedings of the First International Conference on Terrain-Vehicle Systems*, 1961.

25. Bekker, M. G., *Introduction to Terrain-Vehicle Systems*, University of Michigan Press, Ann Arbor, MI, 1969.

26. U. S. Army Engineer Waterways Experiment Station, Technical Memorandum 3-240, 8th Supplement, *Trafficability of Soils, Slope Studies*, May 1951.

27. Rowe, N. C. and Lewis, D. H., "Vehicle Path-Planning in Three Dimensions Using Optics Analogs for Optimizing Visibility and Energy Cost," *Proceedings of the National Aeronautics and Space Administration-Jet Propulsion Laboratory Conference on Space Telerobotics* , February 1989.

28. Kanayama, Y. and DeHaan, G. R., *A Mathematical Theory of Safe Path Planning*, Technical Report, University of California, Santa Barbara, 1988.

29. Dijkstra, E. W., "A Note on Two Problems in Connection with Graphs," *Numer. Math.*, v. 1, 1959.

30. Friedland, P. E., *Knowledge-Based Experiment Design in Molecular Genetics*, Doctoral Dissertation, Rep. No. 79-771, Computer Science Department, Stanford University, 1979.

31. Khatib, O., "Dynamic Control of Manipulators in Operational Space," *Sixth CISM-IFToMM Congress on Theory Of Machines and mechanisms*, New Delhi, India, December 1983.

32. Mitchell, J. S. B., Mount, D. M., and Papadimitriou, C. H., "The Discrete Geodesic Problem," *SIAM Journal of Computing*, v. 16, no. 4, August 1987.

33. Canny, J. F. and Reif, J., "New Lower Bound Techniques for Robot Motion Planning Problems," *Proceedings 28th Annual IEEE Symposium on Foundations of Computer Science*, 1987.

34. Sharir, M. and Schorr, A., "On Shortest Paths in Polyhedral Spaces," *SIAM Journal of Computing*, v. 15, no. 1, February 1986.

35. Mitchell, J. S. B. and Papadimitriou, C. H., *The Weighted Region Problem*, Technical Report (Department of Operations Research), Stanford University, July 1986.

36. Rowe, N. C., *Roads, Rivers, and Rocks: Optimal Two-dimensional Route Planning Around Linear Features for a Mobile Robot*, Technical Report, NPS52-87-027, U. S. Naval Postgraduate School, June 1987.

37. Mitchell, J. S. B. and Kiersey, D. M., "Planning Strategic Paths Through Variable Terrain Data," *SPIE Applications of Artificial Intelligence*, v. 485, 1984.

38. Linden, T. A., Marsh, J. P., and Dove, D. L., "Architecture and Early Experience with Planning for the ALV," *IEEE International Conference on Robotics and Automation*, April 1986.

39. Meriam, J. L. and Kraige, L. G., *Dynamics*, v. 2, J. Wiley and Sons, New York, NY, 1986.

40. Beer, F. P. and Johnston, E. R., *Vector Mechanics for Engineers: Statics and Dynamics*, McGraw-Hill Book Company, New York, NY, 1972.

41. Loomis, L. H., *Calculus*, Addison-Wesley Publishing Company, Reading, MA, 1982.

42. Seely, F. B., et al., *Analytic Mechanics for Engineers*, J. Wiley and Sons, New York, NY, 1958.

43. Gabrielli, G. and Von-Karman, T. H., "What Price Speed," *Mechanical Engineering*, v. 72, no. 10, 1950.

44. McGhee, R. B., et al., *An Approach to Computer Coordination of Motion for Energy-Efficient Walking Machines*, Bulletin of the Mechanical Engineering Laboratory, No. 43, Tsukuba, Ibaraki

Pref., Japan, 1986.

45. Mortenson, M. E., *Geometric Modeling*, J. Wiley and Sons, New York, NY, 1985.
46. Muehrcke, P. C., *Map Use, Reading, Analysis, and Interpretation*, JP Publications, Madison, WI, 1980.
47. McGhee, R. B. and Iswandhi, G. I., "Adaptive Locomotion of a Multilegged Robot over Rough Terrain," *IEEE Transactions, Systems, Man, and Cybernetics*, v. SMC-9, no. 4, April 1979.
48. McGhee, R. B. and Frank, A. A., "On the Stability of Quadruped Creeping Gaits," *Mathematical Biosciences*, v. 3, no. 3, October 1968.
49. Brooks, R. A., "Solving the Find Path Problem by Good Representation of Free Space," *IEEE Transactions on Systems, Man, and Cybernetics*, v. SMC-13, no. 3, March/April 1983.
50. Rowe, N. C. and Ross, R. S., *Optimal Grid-free Path Planning Across Arbitrarily-Contoured Terrain With Anisotropic Friction and Gravity Effects*, Technical Report, NPS52-89-003, U. S. Naval Postgraduate School, November 1988.
51. Reklaitis, G. V., Ravindran, A., and Ragsdell, K. M., *Engineering Optimization Methods and Applications*, John Wiley and Sons, 1983.
52. Steele, G. L., *Common LISP: The Language*, Digital Press, Hanover, MA, 1984.
53. Rogers, D. F., *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, NY, 1985.
54. Department of the Army Technical Manual, TM 9-2300-257-10, *Operator's Manual, M113-APC Armored Personnel Carrier*, February 1973.
55. Department of the Army Technical Manual, TM 9-2320-260-10, *Operator's Manual, M813-CT Cargo Truck*, June 1985.
56. Department of the Army Technical Manual, TM 9-2320-280-10, *Operator's Manual, M966-ATC Armored Tow Carrier*, April 1985.

APPENDIX A - LISP SOURCE CODE FOR PROGRAM

```
;*****************************************************************************
;
; File: MPP-LISP-LIBRARY
;
; Functions: SQR (number)
;            VECTOR-ADD (vector1 vector2)
;            VECTOR-SUB (vector1 vector2)
;            VECTOR-MAGNITUDE (vector)
;            VECTOR-SCALE (scalar vector)
;            VECTOR-PROJECT (vector1 vector2)
;            DOT-PRODUCT (vector1 vector2)
;            CROSS-PRODUCT (vector1 vector2)
;            POINT-EQUAL-P (x1 y1 x2 y2)
;            LINE-LENGTH (x1 y1 x2 y2)
;            LINE-EQUATION (x1 y1 x2 y2)
;            LINE-EQUATION-SOLUTION (line-equation x y)
;            LINE-SEGMENT-RANGE-P (x1 y1 x2 y2 x3 y3)
;            LINE-INTERSECTION (line-equation1 line-equation2)
;            LINE-MIDPOINT (x1 y1 x2 y2)
;            SPLIT-LINE (x1 y1 x2 y2)
;            INTERSECT-COLLINEAR (x1 y1 x2 y2 x3 y3 x4 y4)
;            INTERSECT (x1 y1 x2 y2 x3 y3 x4 y4)
;            POINT-OFFSET (x1 y1 x2 y2 offset)
;            CONCAT (&rest args)
;            REMOVE-ITEMS (list1 list2)
;            ROTATE-LEFT (list)
;            ROTATE-RIGHT (list)
;            LIST-LENGTH-1 (list)
;            EQUAL-WITHIN-TOLERANCE (number1 number2 tolerance)
;            MAKE-SIGNIFICANT-FIGURES (number significant-figures)
;            DEGREES-TO-RADIANS (degrees)
;            RADIANS-TO-DEGREES (radians)
;
;*****************************************************************************


(defun sqr (number)
  (* number number))


(defun vector-add (vector1 vector2)
  (mapcar '+ vector1 vector2))


(defun vector-sub (vector1 vector2)
  (mapcar '- vector1 vector2))


(defun vector-magnitude (vector)
  (sqrt (apply '+ (mapcar 'sqr vector))))


(defun vector-scale (scalar vector)
  (let ((result nil))
    (dolist (vector1 vector result)
      (setf result (cons (* scalar vector1) result)))
    (reverse result)))
```

```
(defun vector-project (vector1 vector2)
  (let* ((vsmagnitude (sqr (vector-magnitude vector2)))
         (xproject (/ (dot-product vector1 vector2) vsmagnitude))
         (vscale (vector-scale xproject vector2)))
    (if (and (>= xproject 0.0) (<= xproject 1.0)) vscale nil)))


(defun dot-product (vector1 vector2)
  (apply '+ (mapcar '* vector1 vector2)))


(defun cross-product (vector1 vector2)
  (let* ((x1 (first vector1))
         (x2 (second vector1))
         (x3 (third vector1))
         (y1 (first vector2))
         (y2 (second vector2))
         (y3 (third vector2))
         (xr (- (* x2 y3) (* x3 y2)))
         (yr (- (* x3 y1) (* x1 y3)))
         (zr (- (* x1 y2) (* x2 y1))))
    (list xr yr zr)))


(defun point-equal-p (x1 y1 x2 y2)
  (if (and (equal-within-tolerance x1 x2 0.01)
           (equal-within-tolerance y1 y2 0.01)) t nil))


(defun line-length (x1 y1 x2 y2)
   (sqrt (+ (expt (- x2 x1) 2) (expt (- y2 y1) 2))))


(defun line-equation (x1 y1 x2 y2)
  (list (- y1 y2) (- x2 x1) (- (* x1 y2) (* y1 x2))))


(defun line-equation-solution (line-equation x y)
  (+ (* (first line-equation) x)
     (* (second line-equation) y)
     (third line-equation)))


(defun line-segment-range-p (x1 y1 x2 y2 x3 y3)
  (let ((xmax (max x2 x3))
        (xmin (min x2 x3))
        (ymax (max y2 y3))
        (ymin (min y2 y3)))
    (cond ((and (or (> x1 xmax) (< x1 xmin))
                (or (> y1 ymax) (< y1 ymin))) nil)
          ((or (> y1 ymax) (< y1 ymin)) nil)
          ((or (> x1 xmax) (< x1 xmin)) nil)
          (t t))))


(defun line-intersection (line-equation1 line-equation2)
  (let* ((a1 (first line-equation1))
         (b1 (second line-equation1))
         (c1 (third line-equation1))
         (a2 (first line-equation2))
```

```
        (b2 (second line-equation2))
        (c2 (third line-equation2))
        (a (- (* b1 c2) (* c1 b2)))
        (b (- (* c1 a2) (* a1 c2)))
        (c (- (* a1 b2) (* b1 a2))))
    (if (= c 0.0) nil (list (/ a c) (/ b c)))))


(defun line-midpoint (x1 y1 x2 y2)
  (list (/ (+ x1 x2) 2.0) (/ (+ y1 y2) 2.0)))


(defun split-line (x1 y1 x2 y2)
  (let* ((midpoint (line-midpoint x1 y1 x2 y2))
         (midpoint-x (first midpoint))
         (midpoint-y (second midpoint)))
    (list (list x1 y1) (list midpoint-x midpoint-y) (list x2 y2))))


(defun intersect-collinear (x1 y1 x2 y2 x3 y3 x4 y4)
  (cond ((and (point-equal-p x1 y1 x3 y3)
              (point-equal-p x2 y2 x4 y4))
         (list x1 y1 x2 y2))
        ((and (point-equal-p x1 y1 x4 y4)
              (point-equal-p x2 y2 x3 y3))
         (list x1 y1 x2 y2))
        ((and (line-segment-range-p x1 y1 x3 y3 x4 y4)
              (line-segment-range-p x2 y2 x3 y3 x4 y4))
         (list x1 y1 x2 y2))
        ((and (line-segment-range-p x3 y3 x1 y1 x2 y2)
              (line-segment-range-p x4 y4 x1 y1 x2 y2))
         (list x3 y3 x4 y4))
        ((and (line-segment-range-p x3 y3 x1 y1 x2 y2)
              (line-segment-range-p x2 y2 x3 y3 x4 y4))
         (list x2 y2 x3 y3))
        ((and (line-segment-range-p x2 y2 x3 y3 x4 y4)
              (line-segment-range-p x4 y4 x1 y1 x2 y2))
         (list x2 y2 x4 y4))
        ((and (line-segment-range-p x1 y1 x3 y3 x4 y4)
              (line-segment-range-p x3 y3 x1 y1 x2 y2))
         (list x1 y1 x3 y3))
        ((and (line-segment-range-p x1 y1 x3 y3 x4 y4)
              (line-segment-range-p x4 y4 x1 y1 x2 y2))
         (list x1 y1 x4 y4))))


(defun intersect (x1 y1 x2 y2 x3 y3 x4 y4)
  (let* ((le1 (line-equation x1 y1 x2 y2))
         (le2 (line-equation x3 y3 x4 y4))
         (s1 (line-equation-solution le2 x1 y1))
         (s2 (line-equation-solution le2 x2 y2))
         (s3 (line-equation-solution le1 x3 y3))
         (s4 (line-equation-solution le1 x4 y4))
         (a1 (first le1))
         (b1 (second le1))
         (c1 (third le1))
         (a2 (first le2))
         (b2 (second le2))
         (c2 (third le2))
         (a (- (* b1 c2) (* c1 b2)))
```

```
        (b (- (* c1 a2) (* a1 c2)))
        (c (- (* a1 b2) (* b1 a2))))
    (cond ((point-equal-p x1 y1 x2 y2) nil)
          ((point-equal-p x3 y3 x4 y4) nil)
          ((and (equal-within-tolerance s1 0.0 0.01)
                (equal-within-tolerance s2 0.0 0.01)
                (equal-within-tolerance s3 0.0 0.01)
                (equal-within-tolerance s4 0.0 0.01))
           (intersect-collinear x1 y1 x2 y2 x3 y3 x4 y4))
          ((and (equal-within-tolerance s1 0.0 0.01)
                (or (equal-within-tolerance s3 0.0 0.01)
                    (equal-within-tolerance s4 0.0 0.01))) (list x1 y1))
          ((and (equal-within-tolerance s2 0.0 0.01)
                (or (equal-within-tolerance s3 0.0 0.01)
                    (equal-within-tolerance s4 0.0 0.01))) (list x2 y2))
          ((or (and (or (equal-within-tolerance s1 0.0 0.01)
                        (equal-within-tolerance s2 0.0 0.01)) (< (* s3 s4) 0.0))
               (and (or (equal-within-tolerance s3 0.0 0.01)
                        (equal-within-tolerance s4 0.0 0.01)) (< (* s1 s2) 0.0))
               (and (< (* s1 s2) 0.0) (< (* s3 s4) 0.0)))
           (list (/ a c) (/ b c)))
          (t nil))))


(defun point-offset (x1 y1 x2 y2 offset)
  (let* ((delta-x (- x2 x1))
         (delta-y (- y2 y1))
         (x-offset (* delta-x offset))
         (y-offset (* delta-y offset)))
    (list (+ x1 x-offset) (+ y1 y-offset))))


(defun concat (&rest args)
  (intern (apply #'concatenate 'simple-string (mapcar '(lambda (x)
  (if (numberp x) (write-to-string x) (string x))) args))))


(defun remove-items (list1 list2)
  (dolist (item list1 list2)
    (setf list2 (remove item list2))))


(defun rotate-left (list)
  (append (rest list) (list (first list))))


(defun rotate-right (list)
  (append (last list) (remove (first (last list)) list)))


(defun list-length-1 (list)
  (if (= (length list) 1) t nil))


(defun equal-within-tolerance (number1 number2 tolerance)
  (if (< (abs (- number1 number2)) tolerance) t nil))


(defun make-significant-figures (number significant-figures)
  (let ((factor (expt 10.0 significant-figures)))
```

```
    (/ (round (* number factor)) factor)))


(defun radians-to-degrees (radians)
  (/ radians 0.0174533))


(defun degrees-to-radians (degrees)
  (* degrees 0.0174533 ))


;*****************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-BUILD-MAP-UTILITIES
;
; Structures: VERTEX (x-coord y-coord z-coord edge-list visibility-list)
;             EDGE (vertex-list adjacency-list visibility-list)
;             REGION (edge-list slope orientation surface-material
;                     surface-condition surface-covering type
;                     stability-constraints braking-constraints)
;
; Functions: BUILD-VERTEX (xcoord ycoord zcoord elist vslist)
;            BUILD-EDGE (vlist adlist vslist)
;            BUILD-REGION (elist rslope rorientation rsmaterial rscondition
;                          rscovering rtype rmresistance rsconstraints
;                          rbconstraints)
;            BUILD-VERTEX-INTERACTIVE ()
;            BUILD-EDGE-INTERACTIVE ()
;            BUILD-REGION-INTERACTIVE ()
;            BUILD-VIRTUAL-VERTEX (xcoord ycoord zcoord edge)
;            BUILD-VIRTUAL-EDGE (vertex1 vertex2 edge)
;            BUILD-VERTEX-VISIBILITY (vertexlist)
;            BUILD-EDGE-VISIBILITY (edgelist)
;            BUILD-BRAKING-CONSTRAINTS ()
;            BUILD-REGION-TYPE ()
;            SAVE-MAP (filename)
;            SAVE-MAP-STATE (filename)
;            SAVE-MAP-LIST (filename)
;            SAVE-STRUCTURE (structure-list filename)
;            LOAD-MAP (filename)
;            LOAD-MAP-STATE (filename)
;            LOAD-MAP-LIST (filename)
;            LOAD-STRUCTURE (filename)
;
; Global Variables: *input-stream*           {global input operations}
;                   *output-stream*          {global output operations}
;                   *vertex-list*            {global vertex list}
;                   *edge-list*              {global edge list}
;                   *region-list*            {global region list}
;                   *virtual-vertex-list*    {global virtual vertex list}
;                   *virtual-edge-list*      {global virtual edge list}
;                   *background-edge-list*   {global edges of background}
;                   *background-region-list* {global subregions of background}
;                   *boundary-vertex-list*   {global background vertices}
;                   *boundary-edge-list*     {global background edges}
;                   *terrain-map-list*       {global list of available maps}
;
;*****************************************************************************


;*****************************************************************************
;
; Geometric Model: Definition of Primitive Structures
;
;*****************************************************************************


(defstruct vertex x-coord y-coord z-coord edge-list visibility-list)


(defstruct edge vertex-list adjacency-list visibility-list)
```

```
(defstruct region edge-list slope orientation surface-material surface-condition
          surface-covering type stability-constraints braking-constraints)
```

```
;*****************************************************************************
;
; Geometric Model: Construction of Primitive Structures
;
;*****************************************************************************


(defun build-vertex (xcoord ycoord zcoord elist vslist)
  (make-vertex ':x-coord xcoord
               ':y-coord ycoord
               ':z-coord zcoord
               ':edge-list elist
               ':visibility-list vslist))


(defun build-edge (vlist adlist vslist)
  (make-edge ':vertex-list vlist
             ':adjacency-list adlist
             ':visibility-list vslist))


(defun build-region (elist rslope rorientation rsmaterial rscondition rscovering
                     rtype rsconstraints rbconstraints)
  (make-region ':edge-list elist
               ':slope rslope
               ':orientation rorientation
               ':surface-material rsmaterial
               ':surface-condition rscondition
               ':surface-covering rscovering
               ':type rtype
               ':stability-constraints rsconstraints
               ':braking-constraints rbconstraints))


;*****************************************************************************
;
; Geometric Model: Interactive Structure Construction
;
;*****************************************************************************


(defun build-vertex-interactive ()
  (let ((designation nil)
        (xcoord nil)
        (ycoord nil)
        (zcoord nil)
        (elist nil)
        (vslist nil))
    (terpri) (terpri)
    (princ "DESIGNATION: ")
    (setf designation (read))
    (princ "X-COORD: ")
    (setf xcoord (read))
    (princ "Y-COORD: ")
    (setf ycoord (read))
    (princ "Z-COORD: ")
    (setf zcoord (read))
    (princ "EDGE-LIST: ")
    (setf elist (read)) (terpri)
    (princ "VISIBILITY-LIST: ")
```

```
    (setf vslist (read)) (terpri)
    (eval (list 'setf designation
                (list 'build-vertex xcoord ycoord zcoord elist vslist)))
    (setf *vertex-list* (cons designation *vertex-list*)) t))


(defun build-edge-interactive ()
  (let ((designation nil)
        (vlist nil)
        (adlist nil)
        (vslist nil))
    (terpri) (terpri)
    (princ "DESIGNATION: ")
    (setf designation (read))
    (princ "VERTEX-LIST: ")
    (setf vlist (read)) (terpri)
    (princ "ADJACENCY-LIST: ")
    (setf adlist (read)) (terpri)
    (princ "VISIBILITY-LIST: ")
    (setf vslist (read)) (terpri)
    (eval (list 'setf designation (list 'build-edge vlist adlist vslist)))
    (setf *edge-list* (cons designation *edge-list*)) t))


(defun build-region-interactive ()
  (let ((designation nil)
        (elist nil)
        (rslope nil)
        (rorientation nil)
        (rsmaterial nil)
        (rscondition nil)
        (rscovering nil)
        (rtype nil)
        (rsconstraints nil)
        (rbconstraints nil))
    (terpri) (terpri)
    (princ "DESIGNATION: ")
    (setf designation (read))
    (princ "EDGE-LIST: ")
    (setf elist (read)) (terpri)
    (princ "SLOPE: ")
    (setf rslope (read))
    (princ "ORIENTATION: ")
    (setf rorientation (read))
    (princ "SURFACE-MATERIAL: ")
    (setf rsmaterial (read))
    (princ "SURFACE-CONDITION: ")
    (setf rscondition (read))
    (princ "SURFACE-COVERING: ")
    (setf rscovering (read))
    (princ "TYPE: ")
    (setf rtype (read))
    (princ "STABILITY-CONSTRAINTS: ")
    (setf rsconstraints (read)) (terpri)
    (princ "BRAKING-CONSTRAINTS: " )
    (setf rbconstraints (read)) (terpri)
    (eval (list 'setf designation (list 'build-region elist rslope rorientation
                                        rsmaterial rscondition rscovering rtype
                                        rsconstraints rbconstraints)))
    (setf *region-list* (cons designation *region-list*)) t))
```

```
;*****************************************************************************
;
; Geometric Model: Construction of Virtual Structures
;
;*****************************************************************************


(defun build-virtual-vertex (xcoord ycoord zcoord edge)
  (let* ((virtual-vertex (concat 'vv- (1+ *virtual-vertex-count*)))
         (region-list (edge-adjacency-list (eval edge)))
         (incident-vertex-list (edge-vertex-list (eval edge)))
         (vertex-list-1 (get-vertexlist-from-region (first region-list)))
         (vertex-list-2 (if (second region-list)
                            (get-vertexlist-from-region (second region-list))))
         (vertex-list (remove-duplicates
                       (remove-items incident-vertex-list
                         (append vertex-list-1 vertex-list-2))))
         (goal-visibility-1
           (if (point-in-region-p (vertex-x-coord (eval 'g-v))
                                  (vertex-y-coord (eval 'g-v))
                                  (first region-list)) t nil))
         (goal-visibility-2
           (if (second region-list)
               (if (point-in-region-p (vertex-x-coord (eval 'g-v))
                                      (vertex-y-coord (eval 'g-v))
                                      (second region-list)) t nil)))
         (visibility-list (if (or goal-visibility-1 goal-visibility-2)
                              (cons 'g-v vertex-list) vertex-list)))
    (eval (list 'setf virtual-vertex
          (build-vertex xcoord ycoord zcoord (list edge) visibility-list)))
    (setf *virtual-vertex-list* (cons virtual-vertex *virtual-vertex-list*))
    (setf *virtual-vertex-count* (1+ *virtual-vertex-count*)) virtual-vertex))


(defun build-virtual-edge (vertex1 vertex2 edge)
  (let ((virtual-edge (concat 've- (1+ *virtual-edge-count*))))
    (eval (list 'setf virtual-edge
          (build-edge (list vertex1 vertex2) (edge-adjacency-list (eval edge))
                      (edge-visibility-list (eval edge)))))
    (setf *virtual-edge-list* (cons virtual-edge *virtual-edge-list*))
    (setf *virtual-edge-count* (1+ *virtual-edge-count*)) virtual-edge))


;*****************************************************************************
;
; Geometric Model: Construction of Visibility Lists
;
;*****************************************************************************


(defun build-vertex-visibility (vertexlist)
  (dolist (vertex vertexlist)
    (setf (vertex-visibility-list (eval vertex))
    (remove-duplicates
      (remove vertex
        (apply 'append (mapcar 'get-vertexlist-from-region
          (remove-if 'obstacle-region-p
            (get-regionlist-from-vertex vertex)))))))) t)
```

```
(defun build-edge-visibility (edgelist)
  (dolist (edge edgelist)
    (let* ((region-list (edge-adjacency-list (eval edge))))
      (if (= (length region-list) 1)
          (setf (edge-visibility-list (eval edge))
                (remove edge (region-edge-list (eval (first region-list)))))
          (if (= (length region-list) 2)
              (let* ((region-1 (first region-list))
                     (region-2 (second region-list))
                     (obstacle-region-1
                       (if (obstacle-region-p region-1) t nil))
                     (obstacle-region-2
                       (if (obstacle-region-p region-2) t nil)))
                (if (and (null obstacle-region-1) obstacle-region-2)
                    (setf (edge-visibility-list (eval edge))
                          (remove edge (region-edge-list (eval region-1))))
                    (if (and obstacle-region-1 (null obstacle-region-2))
                        (setf (edge-visibility-list (eval edge))
                              (remove edge (region-edge-list (eval region-2))))
                        (if (and (null obstacle-region-1)
                                 (null obstacle-region-2))
                            (setf (edge-visibility-list (eval edge))
                                  (remove edge
                                    (apply 'append
                                      (mapcar 'region-edge-list
                                        (mapcar 'eval
                                          region-list)))))))))))))) t)


;*****************************************************************************
;
; Geometric Model: Construction of Vehicle-Dependent Structures
;
;*****************************************************************************


(defun build-braking-constraints ()
  (if *current-vehicle*
      (dolist (region *region-list*)
        (if (anisotropic-region-p region)
            (eval (list 'setf (list 'region-braking-constraints (eval region))
                        (list 'braking-headings *critical-coasting-angle*
                              (region-slope (eval region))
                              (region-orientation (eval region)))))))) t)


(defun build-region-type ()
  (if *current-vehicle*
      (let ()
        (dolist (region *region-list*)
          (if (isotropic-region-p region)
              (setf (region-type (eval region)) 'isotropic)
              (if (anisotropic-safe-region-p region)
                  (setf (region-type (eval region)) 'anisotropic-safe)
                  (if (anisotropic-partially-safe-region-p region)
                      (setf (region-type (eval region))
                        'anisotropic-partially-safe)
                      (if (obstacle-region-p region)
                          (setf (region-type (eval region)) 'obstacle))))))
        (dolist (background-region *background-region-list*)
```

```
          (setf (region-type (eval background-region)) 'isotropic)))) t)

;*****************************************************************************
;
; Geometric Model: Saving and Restoration of Structures
;
;*****************************************************************************


(defun save-map (filename)
  (eval (list 'setf '*output-stream*
    (list 'open filename ':direction ':output)))
  (print *vertex-list* *output-stream*)
  (print *edge-list* *output-stream*)
  (print *background-edge-list* *output-stream*)
  (print *region-list* *output-stream*)
  (print *background-region-list* *output-stream*)
  (dolist (gstructure (append *vertex-list* *edge-list* *background-edge-list*
                       *region-list* *background-region-list*))
    (print (eval gstructure) *output-stream*))
  (close *output-stream*) t)


(defun save-map-state (filename)
  (eval (list 'setf '*output-stream*
    (list 'open filename ':direction ':output)))
  (print *vertex-list* *output-stream*)
  (print *edge-list* *output-stream*)
  (print *background-edge-list* *output-stream*)
  (print *region-list* *output-stream*)
  (print *background-region-list* *output-stream*)
  (dolist (gstructure (append '(s-v g-v) *vertex-list* *edge-list*
                       *background-edge-list* *region-list*
                       *background-region-list*))
    (print (eval gstructure) *output-stream*))
  (close *output-stream*) t)


(defun save-map-list (filename)
  (eval (list 'setf '*output-stream*
    (list 'open filename ':direction ':output)))
  (print *terrain-map-list* *output-stream*)
  (close *output-stream*) t)


(defun save-structure (structure-list filename)
  (eval (list 'setf '*output-stream*
    (list 'open filename ':direction ':output)))
  (princ structure-list *output-stream*)
  (terpri *output-stream*)
  (terpri *output-stream*)
  (dolist (gstructure structure-list)
    (princ (eval gstructure) *output-stream*)
    (terpri *output-stream*)
    (terpri *output-stream*))
  (close *output-stream*) t)


(defun load-map (filename)
```

```
  (eval (list 'setf '*input-stream* (list 'open filename ':direction ':input)))
  (setf *vertex-list* (read *input-stream*))
  (setf *edge-list* (read *input-stream*))
  (setf *background-edge-list* (read *input-stream*))
  (setf *region-list* (read *input-stream*))
  (setf *background-region-list* (read *input-stream*))
  (dolist (gstructure (append *vertex-list* *edge-list* *background-edge-list*
                      *region-list* *background-region-list*))
    (eval (list 'setf gstructure (list 'read '*input-stream*))))
  (close *input-stream*) t)


(defun load-map-state (filename)
  (eval (list 'setf '*input-stream* (list 'open filename ':direction ':input)))
  (setf *vertex-list* (read *input-stream*))
  (setf *edge-list* (read *input-stream*))
  (setf *background-edge-list* (read *input-stream*))
  (setf *region-list* (read *input-stream*))
  (setf *background-region-list* (read *input-stream*))
  (dolist (gstructure (append '(s-v g-v) *vertex-list* *edge-list*
                      *background-edge-list* *region-list*
                      *background-region-list*))
    (eval (list 'setf gstructure (list 'read '*input-stream*))))
  (close *input-stream*) t)


(defun load-map-list (filename)
  (eval (list 'setf '*input-stream* (list 'open filename ':direction ':input)))
  (setf *terrain-map-list* (read *input-stream*))
  (close *input-stream*) t)


(defun load-structure (filename)
  (let ((structure-list nil))
    (eval (list 'setf '*input-stream*
      (list 'open filename ':direction ':input)))
    (setf structure-list (read *input-stream*))
    (dolist (gstructure structure-list)
      (eval (list 'setf gstructure (list 'read '*input-stream*))))
    (close *input-stream*) structure-list))


;*****************************************************************************
;
; Geometric Model: Definition and Initialization of Global Variables
;
;*****************************************************************************


(defvar *input-stream*)
(defvar *output-stream*)
(defvar *vertex-list*)
(defvar *edge-list*)
(defvar *region-list*)
(defvar *virtual-vertex-list*)
(defvar *virtual-edge-list*)
(defvar *background-edge-list*)
(defvar *background-region-list*)
(defvar *boundary-vertex-list*)
(defvar *boundary-edge-list*)
```

```
(defvar *terrain-map-list*)


(setf *input-stream* nil)
(setf *output-stream* nil)
(setf *vertex-list* nil)
(setf *edge-list* nil)
(setf *region-list* nil)
(setf *virtual-vertex-list* nil)
(setf *virtual-edge-list* nil)
(setf *background-edge-list* nil)
(setf *background-region-list* nil)
(setf *boundary-vertex-list* '(v-nw v-ne v-sw v-se))
(setf *boundary-edge-list* '(e-n e-s e-e e-w))
(setf *terrain-map-list* '(synthetic-terrain-map))


;*******************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-BUILD-VEHICLE-UTILITIES
;
; Structures: VEHICLE (name type weight center-of-gravity coasting-slope
;                      contour-slope gradient-slope stability-safety-margin)
;
; Functions: BUILD-VEHICLE (vname vtype vweight vcg vcslope vctslope vgslope
;                           vsmargin)
;            BUILD-VEHICLE-INTERACTIVE ()
;            SAVE-VEHICLES (filename)
;            LOAD-VEHICLES (filename)
;
; Global Variables: *vehicle-list* {global vehicle concepts}
;
;*****************************************************************************


;*****************************************************************************
;
; Vehicle Model: Definition of Primitive Structures
;
;*****************************************************************************


(defstruct vehicle name type weight center-of-gravity coasting-slope
           contour-slope gradient-slope stability-safety-margin)


;*****************************************************************************
;
; Vehicle Model: Construction of Primitive Structures
;
;*****************************************************************************


(defun build-vehicle (vname vtype vweight vcg vcslope vctslope vgslope vsmargin)
  (make-vehicle ':name vname
                ':type vtype
                ':weight vweight
                ':center-of-gravity vcg
                ':coasting-slope vcslope
                ':contour-slope vctslope
                ':gradient-slope vgslope
                ':stability-safety-margin vsmargin))


;*****************************************************************************
;
; Vehicle Model: Interactive Structure Construction
;
;*****************************************************************************


(defun build-vehicle-interactive ()
  (let ((designation nil)
        (vname nil)
        (vtype nil)
        (vweight nil)
        (vcg nil)
```

```
        (vcslope nil)
        (vctslope nil)
        (vgslope nil)
        (vsmargin nil))
    (terpri) (terpri)
    (princ "DESIGNATION: ")
    (setf designation (read))
    (princ "NAME: ")
    (setf vname (read))
    (princ "TYPE: ")
    (setf vtype (read))
    (princ "WEIGHT: ")
    (setf vweight (read))
    (princ "CENTER OF GRAVITY: ")
    (setf vcg (read))
    (princ "COASTING SLOPE: ")
    (setf vcslope (read))
    (princ "CONTOUR SLOPE: ")
    (setf vctslope (read))
    (princ "GRADIENT SLOPE: ")
    (setf vgslope (read))
    (princ "STABILITY-SAFETY-MARGIN: ")
    (setf vsmargin (read))
    (eval (list 'setf designation (list 'build-vehicle vname vtype vweight vcg
                                        vcslope vctslope vgslope vsmargin)))
    (setf *vehicle-list* (cons designation *vehicle-list*)) t))


;*****************************************************************************
;
; Vehicle Model: Saving and Restoration of Structures
;
;*****************************************************************************


(defun save-vehicles (filename)
  (eval (list 'setf '*output-stream*
    (list 'open filename ':direction ':output)))
  (print *vehicle-list* *output-stream*)
  (dolist (gvstructure *vehicle-list*)
    (print (eval gvstructure) *output-stream*))
  (close *output-stream*) t)


(defun load-vehicles (filename)
  (eval (list 'setf '*input-stream* (list 'open filename ':direction ':input)))
  (setf *vehicle-list* (read *input-stream*))
  (dolist (gvstructure *vehicle-list*)
    (eval (list 'setf gvstructure (list 'read '*input-stream*))))
  (close *input-stream*) t)
```

```
;*****************************************************************************
;
; Vehicle Model: Definition and Initialization of Global Variables
;
;*****************************************************************************

(defvar *vehicle-list*)

(setf *vehicle-list* nil)

;*****************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-SPATIAL-REASONING-UTILITIES-I
;
; Functions: GET-COORD-FROM-VERTEX (vertex)
;            GET-XYZ-COORD-FROM-VERTEX (vertex)
;            GET-ELEVATION-FROM-VERTEX (vertex)
;            GET-ELEVATION-FROM-EDGE-POINT (intersection-point edge-point1
;                                           edge-point2)
;            GET-ELEVATION-FROM-INTERIOR-POINT (interior-point region)
;            GET-VERTEX-FROM-POINT (xcoord ycoord)
;            GET-EDGE-FROM-POINT (xcoord ycoord)
;            GET-REGION-FROM-POINT (xcoord ycoord)
;            GET-VERTEXLIST-FROM-REGION (region)
;            GET-VERTEX-LISTS-FROM-EDGELIST (edgelist)
;            GET-VERTICES-AND-EDGES-FROM-REGION (region)
;            GET-REGIONLIST-FROM-VERTEX (vertex)
;            GET-REGIONLIST-FROM-EDGE (edge)
;            GET-INCIDENT-VERTEXLIST-FROM-VERTEX (vertex)
;            GET-INCIDENT-EDGELIST-FROM-EDGE (edge)
;            GET-NON-INCIDENT-VERTEX-FROM-REGION (edge region)
;            GET-EDGELIST-FROM-INTEDGELIST (list)
;            GET-TRAVERSAL-REGION-EDGE-EDGE (edge1 edge2)
;            GET-TRAVERSAL-REGION-VERTEX-VERTEX (vertex1 vertex2)
;            GET-TRAVERSAL-REGION-EDGE-VERTEX (edge vertex)
;            GET-TRAVERSAL-REGION-VERTEX-EDGE (vertex edge)
;            GET-STATIC-EDGE-FROM-VIRTUAL-EDGE (virtual-edge)
;            GET-STATIC-VERTEXLIST-FROM-EDGELIST (edgelist)
;            SELECT-REGION (edge)
;            OBSCURE-EDGE (edge1 edge2)
;            XY-LISTS-FROM-VERTEXLIST (vertexlist)
;            MEMBER-COMMON-VERTEX-LIST (list1 list2)
;            SEQUENCE-VERTEX-LISTS (list)
;            COALESCE-VERTEX-LISTS (list)
;            TRI-EDGE-INCIDENT-VERTEX-P (edge1 edge2 edge3)
;            TRI-EDGE-INCIDENT-EDGE-P (edge1 edge2 edge3)
;            VERTEX-WINDOW-P (window)
;            EDGE-WINDOW-P (window)
;            BOUNDARY-VERTEX-P (vertex)
;            BOUNDARY-EDGE-P (edge)
;            OBSTACLE-VERTEX-P (vertex)
;            OBSTACLE-EDGE-P (edge)
;            OBSCURE-EDGE-P (edge1 edge2)
;            VERTEX-EQUAL-P (vertex1 vertex2)
;            EDGE-EQUAL-P (edge1 edge2)
;            CONVEX-REGION-P (region)
;            ISOTROPIC-REGION-P (region)
;            ANISOTROPIC-REGION-P (region)
;            ANISOTROPIC-SAFE-REGION-P (region)
;            ANISOTROPIC-PARTIALLY-SAFE-REGION-P (region)
;            OBSTACLE-REGION-P (region)
;            UPHILL-P (heading region)
;            DOWNHILL-P (heading region)
;            POINT-AT-VERTEX-P (xcoord ycoord vertex)
;            POINT-ON-EDGE-P (xcoord ycoord edge)
;            POINT-IN-REGION-P (xcoord ycoord region)
;            POINT-IN-REGION-INCLUSIVE-P (xcoord ycoord region)
;
;*****************************************************************************
```

```
;*****************************************************************************
;
; Geometric Model: Structure Manipulation and Interpretation
;
;*****************************************************************************


(defun get-coord-from-vertex (vertex)
  (if (vertex-p (eval vertex))
      (list (vertex-x-coord (eval vertex))
            (vertex-y-coord (eval vertex)))))


(defun get-xyz-coord-from-vertex (vertex)
  (if (vertex-p (eval vertex))
      (list (vertex-x-coord (eval vertex))
            (vertex-y-coord (eval vertex))
            (vertex-z-coord (eval vertex)))))


(defun get-elevation-from-vertex (vertex)
  (if (vertex-p (eval vertex))
      (vertex-z-coord (eval vertex))))


(defun get-elevation-from-edge-point (intersection-point edge-point1
                                      edge-point2)
  (if (point-equal-p (first edge-point1) (second edge-point1)
                     (first edge-point2) (second edge-point2))
                     (third edge-point1)
      (let* ((edge-pt1-x (first edge-point1))
             (edge-pt1-y (second edge-point1))
             (edge-pt1-z (third edge-point1))
             (edge-pt2-x (first edge-point2))
             (edge-pt2-y (second edge-point2))
             (edge-pt2-z (third edge-point2))
             (base-point (if (< edge-pt1-z edge-pt2-z) edge-point1 edge-point2))
             (base-point-x (first base-point))
             (base-point-y (second base-point))
             (base-point-z (third base-point))
             (projected-edge-distance
               (distance-point-to-point edge-pt1-x edge-pt1-y edge-pt2-x
                                        edge-pt2-y))
             (elevation-difference
               (abs (- edge-pt1-z edge-pt2-z))) (edge-slope
                 (radians-to-degrees (atan (/ elevation-difference
                   projected-edge-distance))))
             (intersect-pt-x (first intersection-point))
             (intersect-pt-y (second intersection-point))
             (projected-distance
               (distance-point-to-point base-point-x base-point-y
                 intersect-pt-x intersect-pt-y))
             (offset-elevation
               (* projected-distance (tan (degrees-to-radians edge-slope)))))
        (make-significant-figures (+ base-point-z offset-elevation) 3))))


(defun get-elevation-from-interior-point (interior-point region)
  (let* ((vertex-list (get-vertexlist-from-region region))
         (target-vertex (first vertex-list))
```

```
          (target-vertex-x (vertex-x-coord (eval target-vertex)))
          (target-vertex-y (vertex-y-coord (eval target-vertex)))
          (target-vertex-z (vertex-z-coord (eval target-vertex)))
          (interior-region-slope (region-slope (eval region)))
          (interior-region-orientation (region-orientation (eval region))))
      (if interior-region-orientation
          (let* ((interior-point-x (first interior-point))
                 (interior-point-y (second interior-point))
                 (interior-heading
                   (heading interior-point-x interior-point-y
                            target-vertex-x target-vertex-y))
                 (interior-heading-slope
                   (heading-inclination-angle interior-heading
                     interior-region-slope interior-region-orientation))
                 (projected-interior-distance
                    (distance-point-to-point interior-point-x interior-point-y
                                             target-vertex-x target-vertex-y))
                 (offset-elevation
                   (* projected-interior-distance
                     (tan (degrees-to-radians
                       (abs interior-heading-slope))))))
            (if (plusp interior-heading-slope)
                (make-significant-figures (- target-vertex-z offset-elevation) 3)
                (if (minusp interior-heading-slope)
                    (make-significant-figures
                      (+ target-vertex-z offset-elevation) 3))))
          target-vertex-z)))


(defun get-vertex-from-point (xcoord ycoord)
  (let ((vertex-flag nil)
        (result nil))
    (do ((vertex-list *vertex-list* (rest vertex-list)))
        ((or (null vertex-list) vertex-flag) result)
      (let* ((vertex (first vertex-list))
             (x1 (vertex-x-coord (eval vertex)))
             (y1 (vertex-y-coord (eval vertex))))
        (if (and (= x1 xcoord) (= y1 ycoord))
            (let ()
              (setf vertex-flag t)
              (setf result vertex)))))))


(defun get-edge-from-point (xcoord ycoord)
  (let ((result nil))
    (dolist (edge (append *edge-list* *background-edge-list*) result)
      (if (point-on-edge-p xcoord ycoord edge)
          (setf result (cons edge result))))))


(defun get-region-from-point (xcoord ycoord)
  (let ((result nil))
    (do* ((region (append *region-list* *background-region-list*) (rest region))
          (current-region (first region) (first region)))
         ((or (null region) result))
      (if (point-in-region-p xcoord ycoord current-region)
          (setf result (list current-region))))
    (if (null result)
        (let* ((edgelist (get-edge-from-point xcoord ycoord)))
          (if edgelist
```

```
            (setf result
              (remove-duplicates
                (apply 'append
                  (mapcar 'get-regionlist-from-edge edgelist))))))) result))


(defun get-vertexlist-from-region (region)
  (remove-duplicates (apply 'append (mapcar 'edge-vertex-list
                     (mapcar 'eval (region-edge-list (eval region)))))))


(defun get-vertex-lists-from-edgelist (edgelist)
  (let ((vertex-lists nil))
    (dolist (elist edgelist (reverse vertex-lists))
      (setf vertex-lists (cons (edge-vertex-list (eval elist)) vertex-lists)))))


(defun get-vertices-and-edges-from-region (region)
  (let* ((edge-list (region-edge-list (eval region)))
         (vertex-list (remove-duplicates
                        (apply 'append (mapcar 'edge-vertex-list
                          (mapcar 'eval edge-list))))))
    (append edge-list vertex-list)))


(defun get-regionlist-from-vertex (vertex)
  (remove-duplicates
    (apply 'append (mapcar 'get-regionlist-from-edge
                           (mapcar 'eval (vertex-edge-list (eval vertex)))))))


(defun get-regionlist-from-edge (edge)
  (edge-adjacency-list (eval edge)))


(defun get-incident-vertexlist-from-vertex (vertex)
  (if (member vertex *virtual-vertex-list*) nil
      (remove vertex
        (remove-duplicates
          (apply 'append (mapcar 'edge-vertex-list
            (mapcar 'eval (vertex-edge-list (eval vertex))))))))))


(defun get-incident-edgelist-from-edge (edge)
  (let* ((static-edge (if (member edge *edge-list*) t nil))
         (background-edge (if (member edge *background-edge-list*) t nil))
         (parent-edge (if (member edge *virtual-edge-list*)
                          (get-static-edge-from-virtual-edge edge) nil))
         (vertex-list (edge-vertex-list s-edge))
         (vertex1 (first vertex-list))
         (vertex2 (second vertex-list))
         (virtual-vertex1 (if (member vertex1 *virtual-vertex-list*) t nil))
         (virtual-vertex2 (if (member vertex2 *virtual-vertex-list*) t nil))
         (result nil))
    (if (or static-edge background-edge)
        (setf result (remove edge
                       (apply 'append (mapcar 'vertex-edge-list
                         (mapcar 'eval vertex-list)))))
        (if (and virtual-vertex1 virtual-vertex2)
            (setf result nil)
```

```
            (let ((static-vertex (if virtual-vertex1 vertex2 vertex1)))
              (setf result
                (remove parent-edge
                  (vertex-edge-list (eval static-vertex)))))))) result))


(defun get-non-incident-vertex-from-region (edge region)
  (let ((region-vertex-list (get-vertexlist-from-region region))
        (edge-vertexlist (edge-vertex-list (eval edge))))
    (first (remove-items edge-vertexlist region-vertex-list))))


(defun get-edgelist-from-intedgelist (list)
  (let ((result nil))
    (dolist (intelist list)
      (setf result (append (mapcar 'first intelist) result)))
    (remove-duplicates result)))


(defun get-traversal-region-edge-edge (edge1 edge2)
  (let ((region (first (intersection (edge-adjacency-list (eval edge1))
                                     (edge-adjacency-list (eval edge2))))))
    (if (obstacle-region-p region) nil region)))


(defun get-traversal-region-vertex-vertex (vertex1 vertex2)
  (if (or (equal vertex1 's-v) (equal vertex2 's-v))
      (first (get-region-from-point (vertex-x-coord (eval 's-v))
                                    (vertex-y-coord (eval 's-v))))
      (if (or (equal vertex1 'g-v) (equal vertex2 'g-v))
          (first (get-region-from-point (vertex-x-coord (eval 'g-v))
                                        (vertex-y-coord (eval 'g-v))))
          (let ((edge
                  (first (intersection (vertex-edge-list (eval vertex1))
                                       (vertex-edge-list (eval vertex2))))))
            (if edge
                (select-region edge)
                (let ((region (first (intersection
                                (get-regionlist-from-vertex vertex1)
                                (get-regionlist-from-vertex vertex2)))))
                  (if (obstacle-region-p region) nil region)))))))


(defun get-traversal-region-edge-vertex (edge vertex)
  (if (equal vertex 'g-v)
      (first (get-region-from-point (vertex-x-coord (eval 'g-v))
                                    (vertex-y-coord (eval 'g-v))))
      (let ((region (first (intersection (get-regionlist-from-vertex vertex)
                                         (edge-adjacency-list (eval edge))))))
        (if (obstacle-region-p region) nil region))))


(defun get-traversal-region-vertex-edge (vertex edge)
  (if (equal vertex 's-v)
      (first (get-region-from-point (vertex-x-coord (eval 's-v))
                                    (vertex-y-coord (eval 's-v))))
      (let ((region (first (intersection (get-regionlist-from-vertex vertex)
                                         (edge-adjacency-list (eval edge))))))
        (if (obstacle-region-p region) nil region))))
```

```
(defun get-static-edge-from-virtual-edge (virtual-edge)
  (let ((region-list (edge-adjacency-list (eval virtual-edge))))
    (first (intersection (region-edge-list (eval (first region-list)))
                         (region-edge-list (eval (second region-list)))))))


(defun get-static-vertexlist-from-edgelist (edgelist)
  (let ((static-vertex-list nil))
    (dolist (edge edgelist)
      (let* ((vertex-list (edge-vertex-list (eval edge)))
             (vertex1 (first vertex-list))
             (vertex2 (second vertex-list)))
        (if (member vertex1 *vertex-list*)
            (setf static-vertex-list (cons vertex1 static-vertex-list)))
        (if (member vertex2 *vertex-list*)
            (setf static-vertex-list (cons vertex2 static-vertex-list)))))
    (remove-duplicates static-vertex-list)))


(defun select-region (edge)
  (let* ((region-list (edge-adjacency-list (eval edge)))
         (region-1 (first region-list))
         (region-2 (second region-list))
         (isotropic-region-1 (isotropic-region-p region-1))
         (isotropic-region-2 (isotropic-region-p region-2))
         (anisotropic-safe-region-1 (anisotropic-region-p region-1))
         (anisotropic-safe-region-2 (anisotropic-region-p region-2))
         (anisotropic-partially-safe-region-1
         (anisotropic-partially-safe-region-p region-1))
         (anisotropic-partially-safe-region-2
           (anisotropic-partially-safe-region-p region-2))
         (obstacle-region-1 (obstacle-region-p region-1))
         (obstacle-region-2 (obstacle-region-p region-2))
         (region-1-slope (region-slope (eval region-1)))
         (region-2-slope (region-slope (eval region-2))))
    (cond ((and obstacle-region-1 isotropic-region-2) region-2)
          ((and obstacle-region-2 isotropic-region-1) region-1)
          ((and obstacle-region-1 anisotropic-safe-region-2) region-2)
          ((and obstacle-region-2 anisotropic-safe-region-1) region-1)
          ((and isotropic-region-1 isotropic-region-2) region-1)
          ((and anisotropic-safe-region-1 anisotropic-safe-region-2
           (<= region-1-slope region-2-slope)) region-1)
          ((and anisotropic-safe-region-1 anisotropic-safe-region-2
           (> region-1-slope region-2-slope)) region-2)
          ((and isotropic-region-1 anisotropic-safe-region-2) region-1)
          ((and isotropic-region-2 anisotropic-safe-region-1) region-2)
          ((and isotropic-region-1 anisotropic-partially-safe-region-2)
            region-1)
          ((and isotropic-region-2 anisotropic-partially-safe-region-1)
            region-2)
          ((and anisotropic-safe-region-1 anisotropic-partially-safe-region-2)
            region-1)
          ((and anisotropic-safe-region-2 anisotropic-partially-safe-region-1)
            region-2)
          ((and anisotropic-partially-safe-region-1
                anisotropic-partially-safe-region-2)
           (let* ((vertex-list (edge-vertex-list (eval edge)))
                  (vertex1 (first vertex-list))
                  (vertex2 (second vertex-list))
                  (x-v1 (vertex-x-coord (eval vertex1)))
```

```
                    (y-v1 (vertex-y-coord (eval vertex1)))
                    (x-v2 (vertex-x-coord (eval vertex2)))
                    (y-v2 (vertex-y-coord (eval vertex2)))
                    (edge-heading (heading x-v1 y-v1 x-v2 y-v2))
                    (stability-constraints1
                      (region-stability-constraints (eval region-1)))
                    (stability-constraints2
                      (region-stability-constraints (eval region-2)))
                    (heading-1-lower (heading-range-p edge-heading
                      (first stability-constraints1)
                      (second stability-constraints1)))
                    (heading-1-upper (heading-range-p edge-heading
                      (third stability-constraints1)
                      (fourth  stability-constraints1)))
                    (heading-2-lower (heading-range-p edge-heading
                      (first stability-constraints2)
                      (second stability-constraints2)))
                    (heading-2-upper (heading-range-p edge-heading
                      (third stability-constraints2)
                      (fourth stability-constraints2))))
               (cond ((and (or heading-1-lower heading-1-upper)
                           (or heading-2-lower heading-2-upper))
                      (if (<= region-1-slope region-2-slope) region-1 region-2))
                     ((and (or heading-1-lower heading-1-upper)
                           (or (null heading-2-lower) (null heading-2-upper)))
                      region-1)
                     ((and (or (null heading-1-lower) (null heading-1-upper))
                           (or heading-2-lower heading-2-upper))
                      region-2)
                     ((and (or (null heading-1-lower) (null heading-1-upper))
                           (or (null heading-2-lower) (null heading-2-upper)))
                      nil))))
          ((or (and obstacle-region-1 anisotropic-partially-safe-region-2)
               (and obstacle-region-2 anisotropic-partially-safe-region-1))
           (let* ((vertex-list (edge-vertex-list (eval edge)))
                  (vertex1 (first vertex-list))
                  (vertex2 (second vertex-list))
                  (x-v1 (vertex-x-coord (eval vertex1)))
                  (y-v1 (vertex-y-coord (eval vertex1)))
                  (x-v2 (vertex-x-coord (eval vertex2)))
                  (y-v2 (vertex-y-coord (eval vertex2)))
                  (non-obstacle-region
                    (if obstacle-region-1 region-2 region-1))
                  (edge-heading (heading x-v1 y-v1 x-v2 y-v2))
                  (stability-constraints (region-stability-constraints
                    (eval non-obstacle-region)))
                  (heading-lower (heading-range-p edge-heading
                                   (first stability-constraints)
                                   (second stability-constraints)))
                  (heading-upper (heading-range-p edge-heading
                                   (third stability-constraints)
                                   (fourth stability-constraints))))
             (if (or heading-lower heading-upper)
                 non-obstacle-region nil)))
          (t nil))))


(defun obscure-edge-p (edge1 edge2)
  (let* ((edge1-vertexlist (edge-vertex-list (eval edge1)))
         (edge1-v1 (first edge1-vertexlist))
```

```
        (edge1-v2 (second edge1-vertexlist))
        (edge1-v1-x (vertex-x-coord (eval edge1-v1)))
        (edge1-v1-y (vertex-y-coord (eval edge1-v1)))
        (edge1-v2-x (vertex-x-coord (eval edge1-v2)))
        (edge1-v2-y (vertex-y-coord (eval edge1-v2)))
        (edge1-line-equation
          (line-equation edge1-v1-x edge1-v1-y edge1-v2-x edge1-v2-y))
        (edge2-vertexlist (edge-vertex-list (eval edge2)))
        (edge2-v1 (first edge2-vertexlist))
        (edge2-v2 (second edge2-vertexlist))
        (edge2-v1-x (vertex-x-coord (eval edge2-v1)))
        (edge2-v1-y (vertex-y-coord (eval edge2-v1)))
        (edge2-v2-x (vertex-x-coord (eval edge2-v2)))
        (edge2-v2-y (vertex-y-coord (eval edge2-v2)))
        (edge2-line-equation
          (line-equation edge2-v1-x edge2-v1-y edge2-v2-x edge2-v2-y))
        (line-intersection-point
          (line-intersection edge1-line-equation edge2-line-equation)))
    (if line-intersection-point t nil)))


(defun obscure-edge (edge1 edge2)
  (let* ((edge1-vertexlist (edge-vertex-list (eval edge1)))
         (edge1-v1 (first edge1-vertexlist))
         (edge1-v2 (second edge1-vertexlist))
         (edge1-v1-x (vertex-x-coord (eval edge1-v1)))
         (edge1-v1-y (vertex-y-coord (eval edge1-v1)))
         (edge1-v2-x (vertex-x-coord (eval edge1-v2)))
         (edge1-v2-y (vertex-y-coord (eval edge1-v2)))
         (edge1-line-equation
           (line-equation edge1-v1-x edge1-v1-y edge1-v2-x edge1-v2-y))
         (edge2-vertexlist (edge-vertex-list (eval edge2)))
         (edge2-v1 (first edge2-vertexlist))
         (edge2-v2 (second edge2-vertexlist))
         (edge2-v1-x (vertex-x-coord (eval edge2-v1)))
         (edge2-v1-y (vertex-y-coord (eval edge2-v1)))
         (edge2-v2-x (vertex-x-coord (eval edge2-v2)))
         (edge2-v2-y (vertex-y-coord (eval edge2-v2)))
         (edge2-line-equation
           (line-equation edge2-v1-x edge2-v1-y edge2-v2-x edge2-v2-y))
         (intersection-point
           (line-intersection edge1-line-equation edge2-line-equation)))
    (if intersection-point
        (let ((intersection-pt-x (first intersection-point))
              (intersection-pt-y (second intersection-point))
              (tolerance 0.01))
          (if (or (and (equal-within-tolerance edge1-v1-x
                         intersection-pt-x tolerance)
                       (equal-within-tolerance edge1-v1-y
                         intersection-pt-y tolerance))
                  (and (equal-within-tolerance edge1-v2-x
                         intersection-pt-x tolerance)
                       (equal-within-tolerance edge1-v2-y
                         intersection-pt-y tolerance))
                  (and (equal-within-tolerance edge2-v1-x
                         intersection-pt-x tolerance)
                       (equal-within-tolerance edge2-v1-y
                         intersection-pt-y tolerance))
                  (and (equal-within-tolerance edge2-v2-x
                         intersection-pt-x tolerance)
```

```
                    (equal-within-tolerance edge2-v2-y
                      intersection-pt-y tolerance))) nil
                (let ((in-range-edge1
                       (line-segment-range-p intersection-pt-x intersection-pt-y
                        edge1-v1-x edge1-v1-y edge1-v2-x edge1-v2-y))
                      (in-range-edge2
                       (line-segment-range-p intersection-pt-x intersection-pt-y
                        edge2-v1-x edge2-v1-y edge2-v2-x edge2-v2-y)))
                  (if in-range-edge1 edge1
                      (if in-range-edge2 edge2)))))))


(defun xy-lists-from-vertexlist (vertexlist)
  (let* ((xlist nil)
         (ylist nil))
    (dolist (vertex vertexlist)
      (let ((svertex (eval vertex)))
        (setf xlist (cons (vertex-x-coord svertex) xlist))
        (setf ylist (cons (vertex-y-coord svertex) ylist))))
    (list (reverse xlist) (reverse ylist))))


(defun member-common-vertex-list (list1 list2)
  (let ((vertex1 (first list1))
        (vertex2 (second list1)))
    (if (null (member vertex1 list2)) vertex2 vertex1)))


(defun sequence-vertex-lists (list)
  (let ((vertex-lists nil))
    (do* ((vlists list (rest vlists))
          (vlist1 (first vlists) (first vlists))
          (vlist2 (second vlists) (second vlists))
          (count (- (length list) 1) (- count 1))
          (cmember (member-common-vertex-list vlist1 vlist2)
          (member-common-vertex-list vlist1 vlist2)))
         ((null (rest vlists)) (reverse vertex-lists))
      (if (equal cmember (first vlist1))
          (setf vertex-lists (cons (reverse vlist1) vertex-lists))
          (setf vertex-lists (cons vlist1 vertex-lists)))
      (if (equal cmember (second vlist2))
          (setf vlist2 (reverse vlist2)))
      (if (= count 1) (setf vertex-lists (cons vlist2 vertex-lists))))))


(defun coalesce-vertex-lists (list)
  (let ((vertexlist nil))
    (dolist (vlist list)
      (setf vertexlist (append vertexlist vlist)))
    (remove-duplicates vertexlist)))
```

```
;*****************************************************************************
;
; Geometric Model: Spatial Predicates
;
;*****************************************************************************


(defun tri-edge-incident-vertex-p (edge1 edge2 edge3)
  (let ((edge1-vertexlist (edge-vertex-list (eval edge1)))
        (edge2-vertexlist (edge-vertex-list (eval edge2)))
        (edge3-vertexlist (edge-vertex-list (eval edge3))))
    (if (intersection (intersection edge1-vertexlist edge2-vertexlist)
                      (intersection edge2-vertexlist edge3-vertexlist)) t nil)))


(defun tri-edge-incident-edge-p (edge1 edge2 edge3)
  (let* ((edge1-vertexlist (edge-vertex-list (eval edge1)))
         (edge2-vertexlist (edge-vertex-list (eval edge2)))
         (edge3-vertexlist (edge-vertex-list (eval edge3)))
         (intersection-1
           (first (intersection edge1-vertexlist edge2-vertexlist)))
         (intersection-2
           (first (intersection edge2-vertexlist edge3-vertexlist)))
         (edge2-vertex1 (first edge2-vertexlist))
         (edge2-vertex2 (second edge2-vertexlist)))
    (if (or (and (equal intersection-1 edge2-vertex1)
                 (equal intersection-2 edge2-vertex2))
            (and (equal intersection-1 edge2-vertex2)
                 (equal intersection-2 edge2-vertex1))) t nil)))


(defun vertex-window-p (window)
  (if (vertex-p (eval window)) t nil))


(defun edge-window-p (window)
  (if (edge-p (eval window)) t nil))


(defun boundary-vertex-p (vertex)
  (if (member vertex *boundary-vertex-list*) t nil))


(defun boundary-edge-p (edge)
  (if (member edge *boundary-edge-list*) t nil))


(defun obstacle-vertex-p (vertex)
  (if (member vertex *boundary-vertex-list*) nil
      (let ((edge-list (vertex-edge-list (eval vertex))))
        (if (null (remove-if 'null
                  (mapcar 'obstacle-edge-p edge-list))) nil t))))


(defun obstacle-edge-p (edge)
  (if (member edge *boundary-edge-list*) nil
      (let* ((region-list (edge-adjacency-list (eval edge)))
             (region-1 (first region-list))
             (region-2 (second region-list)))
        (if (or (obstacle-region-p region-1)
```

```
                (obstacle-region-p region-2)) t nil))))


(defun obscure-edge-p (edge1 edge2)
  (let* ((edge1-vertexlist (edge-vertex-list (eval edge1)))
         (edge1-v1 (first edge1-vertexlist))
         (edge1-v2 (second edge1-vertexlist))
         (edge1-v1-x (vertex-x-coord (eval edge1-v1)))
         (edge1-v1-y (vertex-y-coord (eval edge1-v1)))
         (edge1-v2-x (vertex-x-coord (eval edge1-v2)))
         (edge1-v2-y (vertex-y-coord (eval edge1-v2)))
         (edge1-line-equation
         (line-equation edge1-v1-x edge1-v1-y edge1-v2-x edge1-v2-y))
         (edge2-vertexlist (edge-vertex-list (eval edge2)))
         (edge2-v1 (first edge2-vertexlist))
         (edge2-v2 (second edge2-vertexlist))
         (edge2-v1-x (vertex-x-coord (eval edge2-v1)))
         (edge2-v1-y (vertex-y-coord (eval edge2-v1)))
         (edge2-v2-x (vertex-x-coord (eval edge2-v2)))
         (edge2-v2-y (vertex-y-coord (eval edge2-v2)))
         (edge2-line-equation
         (line-equation edge2-v1-x edge2-v1-y edge2-v2-x edge2-v2-y))
         (intersection-point
         (line-intersection edge1-line-equation edge2-line-equation)))
    (if intersection-point
        (let ((intersection-pt-x (first intersection-point))
              (intersection-pt-y (second intersection-point))
              (tolerance 0.01))
          (if (or (and (equal-within-tolerance edge1-v1-x
                         intersection-pt-x tolerance)
                       (equal-within-tolerance edge1-v1-y
                         intersection-pt-y tolerance))
                  (and (equal-within-tolerance edge1-v2-x
                         intersection-pt-x tolerance)
                       (equal-within-tolerance edge1-v2-y
                         intersection-pt-y tolerance))
                  (and (equal-within-tolerance edge2-v1-x
                         intersection-pt-x tolerance)
                       (equal-within-tolerance edge2-v1-y
                         intersection-pt-y tolerance))
                  (and (equal-within-tolerance edge2-v2-x
                         intersection-pt-x tolerance)
                       (equal-within-tolerance edge2-v2-y
                         intersection-pt-y tolerance)))
              nil t)))))


(defun vertex-equal-p (vertex1 vertex2)
  (let* ((s-vertex1 (eval vertex1))
         (s-vertex2 (eval vertex2))
         (x1 (vertex-x-coord s-vertex1))
         (y1 (vertex-y-coord s-vertex1))
         (x2 (vertex-x-coord s-vertex2))
         (y2 (vertex-y-coord s-vertex2)))
    (if (point-equal-p x1 y1 x2 y2) t nil)))


(defun edge-equal-p (edge1 edge2)
  (let* ((s-edge1 (eval edge1))
         (edge1-vertexlist (edge-vertex-list s-edge1))
```

```
        (edge1-v1 (first edge1-vertexlist))
        (edge1-v2 (second edge1-vertexlist))
        (s-edge2 (eval edge2))
        (edge2-vertexlist (edge-vertex-list s-edge2))
        (edge2-v1 (first edge2-vertexlist))
        (edge2-v2 (second edge2-vertexlist)))
    (if (or (and (vertex-equal-p edge1-v1 edge2-v1)
                 (vertex-equal-p edge1-v2 edge2-v2))
            (and (vertex-equal-p edge1-v1 edge2-v2)
                 (vertex-equal-p edge1-v2 edge2-v1))) t nil)))


(defun convex-region-p (region)
  (let* ((s-region (eval region))
         (edge-list (region-edge-list s-region))
         (vertex-list (remove-duplicates
                        (apply 'append
                          (mapcar 'edge-vertex-list (mapcar 'eval edge-list)))))
         (target-edge (first edge-list))
         (s-target-edge (eval target-edge))
         (te-vertex-list (edge-vertex-list s-target-edge))
         (target-vertex1 (first te-vertex-list))
         (s-target-vertex1 (eval target-vertex1))
         (target-vertex2 (second te-vertex-list))
         (s-target-vertex2 (eval target-vertex2))
         (tv1-x (vertex-x-coord s-target-vertex1))
         (tv1-y (vertex-y-coord s-target-vertex1))
         (tv2-x (vertex-x-coord s-target-vertex2))
         (tv2-y (vertex-y-coord s-target-vertex2))
         (edge-line-equation (line-equation tv1-x tv1-y tv2-x tv2-y))
         (plus-or-zero-result nil)
         (minus-or-zero-result nil)
         (result nil)
         (convex-result nil))
    (dolist (vertex vertex-list)
      (let* ((s-vertex (eval vertex))
             (vertex-x (vertex-x-coord s-vertex))
             (vertex-y (vertex-y-coord s-vertex))
             (le-solution
               (line-equation-solution edge-line-equation vertex-x vertex-y)))
        (setf result (cons le-solution result))))
      (setf plus-or-zero-result
        (remove-if 'zerop (remove-if 'plusp result)))
      (setf minus-or-zero-result
        (remove-if 'zerop (remove-if 'minusp result)))
      (setf convex-result (if (or (null plus-or-zero-result)
                                  (null minus-or-zero-result)) t nil)))))


(defun isotropic-region-p (region)
  (if (obstacle-region-p region) nil
      (if (< (region-slope (eval region)) *critical-coasting-angle*) t nil)))


(defun anisotropic-region-p (region)
  (if (obstacle-region-p region) nil
      (if (>= (region-slope (eval region)) *critical-coasting-angle*) t nil)))


(defun anisotropic-safe-region-p (region)
```

```
  (if (obstacle-region-p region) nil
      (let ((slope (region-slope (eval region))))
        (if (and (> slope *critical-coasting-angle*)
                 (< slope *critical-stability-angle*)) t nil))))


(defun anisotropic-partially-safe-region-p (region)
  (if (obstacle-region-p region) nil
      (let ((slope (region-slope (eval region))))
        (if (and (> slope *critical-stability-angle*)
                 (< slope *critical-braking-angle*)) t nil))))


(defun obstacle-region-p (region)
  (if (> (region-slope (eval region)) *critical-braking-angle*) t nil))


(defun uphill-p (heading region)
  (let* ((s-region (eval region))
         (slope (region-slope s-region))
         (orientation (region-orientation s-region))
         (heading-incl-angle
           (heading-inclination-angle heading slope orientation)))
    (if (> heading-incl-angle  0.01) t nil)))


(defun downhill-p (heading region)
  (let* ((s-region (eval region))
         (slope (region-slope s-region))
         (orientation (region-orientation s-region))
         (heading-incl-angle
           (heading-inclination-angle heading slope orientation)))
    (if (< heading-incl-angle -0.01) t nil)))


(defun point-at-vertex (xcoord ycoord vertex)
  (let* ((s-vertex (eval vertex))
         (x1 (vertex-x-coord s-vertex))
         (y1 (vertex-y-coord s-vertex))
         (tolerance 0.01))
    (if (and (equal-within-tolerance xcoord x1 tolerance)
             (equal-within-tolerance ycoord y1 tolerance)) t nil)))


(defun point-on-edge-p (xcoord ycoord edge)
  (let* ((s-edge (eval edge))
         (vertexlist (edge-vertex-list s-edge))
         (s-vertex1 (eval (first vertexlist)))
         (s-vertex2 (eval (second vertexlist)))
         (x1 (vertex-x-coord s-vertex1))
         (y1 (vertex-y-coord s-vertex1))
         (x2 (vertex-x-coord s-vertex2))
         (y2 (vertex-y-coord s-vertex2))
         (le (line-equation x1 y1 x2 y2))
         (le-solution (line-equation-solution le xcoord ycoord))
         (tolerance 0.01)
         (zero-solution
           (if (equal-within-tolerance 0.0 le-solution tolerance) t nil))
         (line-segment-range (if zero-solution
                                 (line-segment-range-p xcoord ycoord
```

```
                                                        x1 y1 x2 y2) nil)))
    (if (and zero-solution line-segment-range) t nil)))


(defun point-in-region-p (xcoord ycoord region)
  (let* ((region-flag nil)
         (edge-list (region-edge-list (eval region)))
         (result-list nil)
         (result-test-positive nil)
         (result-test-negative nil))
    (do* ((vertex-lists (get-vertex-lists-from-edgelist edge-list))
          (svertex-lists (sequence-vertex-lists vertex-lists))
          (edgelist svertex-lists (rest edgelist))
          (current-edge (first edgelist) (first edgelist)))
         ((null edgelist))
      (let* ((s-vertex1 (eval (first current-edge)))
             (s-vertex2 (eval (second current-edge)))
             (x1 (vertex-x-coord s-vertex1))
             (y1 (vertex-y-coord s-vertex1))
             (x2 (vertex-x-coord s-vertex2))
             (y2 (vertex-y-coord s-vertex2))
             (lequation (line-equation x1 y1 x2 y2)))
        (setf result-list
              (cons (line-equation-solution lequation xcoord ycoord)
                    result-list))))
    (dolist (result result-list)
      (if (plusp result)
          (setf result-test-positive (cons t result-test-positive))
          (setf result-test-positive (cons nil result-test-positive))))
    (if (null (remove-if-not 'null result-test-positive))
        (setf region-flag t)
        (let ()
          (dolist (result result-list)
            (if (minusp result)
                (setf result-test-negative (cons t result-test-negative))
                (setf result-test-negative (cons nil result-test-negative))))
            (if (null (remove-if-not 'null result-test-negative))
                (setf region-flag t)))) region-flag))


(defun point-in-region-inclusive-p (xcoord ycoord region)
  (let* ((region-flag nil)
         (tolerance 0.01)
         (edge-list (region-edge-list (eval region)))
         (result-list nil)
         (result-test-positive-or-zero nil)
         (result-test-negative-or-zero nil))
    (do* ((vertex-lists (get-vertex-lists-from-edgelist edge-list))
          (svertex-lists (sequence-vertex-lists vertex-lists))
          (edgelist svertex-lists (rest edgelist))
          (current-edge (first edgelist) (first edgelist)))
         ((null edgelist))
      (let* ((s-vertex1 (eval (first current-edge)))
             (s-vertex2 (eval (second current-edge)))
             (x1 (vertex-x-coord s-vertex1))
             (y1 (vertex-y-coord s-vertex1))
             (x2 (vertex-x-coord s-vertex2))
             (y2 (vertex-y-coord s-vertex2))
             (lequation (line-equation x1 y1 x2 y2)))
        (setf result-list (cons (line-equation-solution lequation xcoord ycoord)
```

```
                                    result-list))))
        (dolist (result result-list)
          (if (or (plusp result)
                  (equal-within-tolerance result 0.0 tolerance))
              (setf result-test-positive-or-zero
                    (cons t result-test-positive-or-zero))
              (setf result-test-positive-or-zero
                    (cons nil result-test-positive-or-zero))))
        (if (null (remove-if-not 'null result-test-positive-or-zero))
            (setf region-flag t)
            (let ()
              (dolist (result result-list)
                (if (or (minusp result)
                        (equal-within-tolerance result 0.0 tolerance))
                    (setf result-test-negative-or-zero
                          (cons t result-test-negative-or-zero))
                    (setf result-test-negative-or-zero
                          (cons nil result-test-negative-or-zero))))
              (if (null (remove-if-not 'null result-test-negative-or-zero))
                  (setf region-flag t)))) region-flag))

;*******************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-SPATIAL-REASONING-UTILITIES-II
;
; Functions: DISTANCE-POINT-TO-POINT (x1 y1 x2 y2)
;            DISTANCE-POINT-TO-EDGE-MIN (xcoord ycoord edge)
;            DISTANCE-POINT-TO-EDGE-MAX (xcoord ycoord edge)
;            DISTANCE-VERTEX-TO-VERTEX (vertex1 vertex2)
;            DISTANCE-VERTEX-TO-EDGE-MIN (vertex edge)
;            DISTANCE-VERTEX-TO-EDGE-MAX (vertex edge)
;            DISTANCE-EDGE-TO-EDGE-MIN (edge1 edge2)
;            DISTANCE-EDGE-TO-EDGE-MAX (edge1 edge2)
;            DISTANCE-WINDOW-TO-WINDOW-LOWER (window1 window2)
;            DISTANCE-WINDOW-TO-WINDOW-UPPER (window1 window2)
;            QUADRANT (x1 y1 x2 y2)
;            HEADING-QUADRANT (heading)
;            QUAD1-HEADING (x1 y1 x2 y2)
;            HEADING (x1 y1 x2 y2)
;            REVERSE-HEADING (heading)
;            NORMALIZE-HEADING (heading)
;            ORDER-HEADINGS (heading1 heading2)
;            INTERVAL-ORDER-HEADINGS (heading-list-1 heading-list-2)
;            HEADING-INCLINATION-ANGLE (heading slope orientation)
;            HEADING-POINT (x1 y1 vheading)
;            HEADING-EQUAL-P (heading1 heading2)
;            HEADING-RANGE-P (heading heading-1 heading-2)
;            INTERVAL-HEADING-RANGE-P (heading-list heading-list-1
;                                      heading-list-2)
;            HEADING-RANGE-INTERSECTION (heading-range1 heading-range2)
;            INTERVAL-HEADING-RANGE-INTERSECTION heading-range-1 heading-range-2)
;            SAME-QUADRANT-P (heading1 heading2)
;            TOTAL-HEADING-RANGE-P (heading heading-11 heading-12 heading-21
;                                   heading-22)
;            BRAKING-HEADING-P (heading region)
;            STABILITY-HEADING-P (heading region)
;            BRAKING-HEADINGS (critical-coasting-angle slope orientation)
;            STABILITY-HEADINGS (stability-offset region)
;            GRADIENT-HEADING (region)
;            CONTOUR-HEADING (region)
;
;*****************************************************************************


;*****************************************************************************
;
; Geometric Model: Distance Functions
;
;*****************************************************************************


(defun distance-point-to-point (x1 y1 x2 y2)
  (line-length x1 y1 x2 y2))


(defun distance-point-to-edge-min (xcoord ycoord edge)
  (let* ((s-edge (eval edge))
         (vlist (edge-vertex-list s-edge))
         (s-vertex1 (eval (first vlist)))
         (s-vertex2 (eval (second vlist)))
         (x1 (vertex-x-coord s-vertex1))
```

```
        (y1 (vertex-y-coord s-vertex1))
        (x2 (vertex-x-coord s-vertex2))
        (y2 (vertex-y-coord s-vertex2))
        (lequation (line-equation x1 y1 x2 y2))
        (a (first lequation))
        (b (second lequation))
        (c (third lequation))
        (distance1 (line-length x1 y1 xcoord ycoord))
        (distance2 (line-length x2 y2 xcoord ycoord))
     (distance3 (/ (abs (+ (* a xcoord) (* b ycoord) c))
                   (sqrt (+ (* a a) (* b b)))))
        (vector1 (list (- xcoord x1) (- ycoord y1)))
        (vector2 (list (- x2 x1) (- y2 y1))))
    (if (null (vector-project vector1 vector2))
        (min distance1 distance2)
        (min distance1 distance2 distance3))))


(defun distance-point-to-edge-max (xcoord ycoord edge)
  (let* ((s-edge (eval edge))
        (vlist (edge-vertex-list s-edge))
        (s-vertex1 (eval (first vlist)))
        (s-vertex2 (eval (second vlist)))
        (x1 (vertex-x-coord s-vertex1))
        (y1 (vertex-y-coord s-vertex1))
        (x2 (vertex-x-coord s-vertex2))
        (y2 (vertex-y-coord s-vertex2))
        (lequation (line-equation x1 y1 x2 y2))
        (a (first lequation))
        (b (second lequation))
        (c (third lequation))
        (distance1 (line-length x1 y1 xcoord ycoord))
        (distance2 (line-length x2 y2 xcoord ycoord))
     (distance3 (/ (abs (+ (* a xcoord) (* b ycoord) c))
                   (sqrt (+ (* a a) (* b b)))))
        (vector1 (list (- xcoord x1) (- ycoord y1)))
        (vector2 (list (- x2 x1) (- y2 y1))))
    (if (null (vector-project vector1 vector2))
        (max distance1 distance2)
        (max distance1 distance2 distance3))))


(defun distance-vertex-to-vertex (vertex1 vertex2)
  (let* ((s-vertex1 (eval vertex1))
        (s-vertex2 (eval vertex2))
        (x1 (vertex-x-coord s-vertex1))
        (y1 (vertex-y-coord s-vertex1))
        (x2 (vertex-x-coord s-vertex2))
        (y2 (vertex-y-coord s-vertex2)))
    (line-length x1 y1 x2 y2)))


(defun distance-vertex-to-edge-min (vertex edge)
  (let* ((s-vertex (eval vertex))
        (s-edge (eval edge))
        (x3 (vertex-x-coord s-vertex))
        (y3 (vertex-y-coord s-vertex))
        (vlist (edge-vertex-list s-edge))
        (s-vertex1 (eval (first vlist)))
        (s-vertex2 (eval (second vlist)))
```

```
        (x1 (vertex-x-coord s-vertex1))
        (y1 (vertex-y-coord s-vertex1))
        (x2 (vertex-x-coord s-vertex2))
        (y2 (vertex-y-coord s-vertex2))
        (lequation (line-equation x1 y1 x2 y2))
        (a (first lequation))
        (b (second lequation))
        (c (third lequation))
        (distance1 (line-length x1 y1 x3 y3))
        (distance2 (line-length x2 y2 x3 y3))
        (distance3 (/ (abs (+ (* a x3) (* b y3) c))
                      (sqrt (+ (* a a) (* b b)))))
        (vector1 (list (- x3 x1) (- y3 y1)))
        (vector2 (list (- x2 x1) (- y2 y1))))
    (if (null (vector-project vector1 vector2))
        (min distance1 distance2)
        (min distance1 distance2 distance3))))


(defun distance-vertex-to-edge-max (vertex edge)
  (let* ((s-vertex (eval vertex))
         (s-edge (eval edge))
         (x3 (vertex-x-coord s-vertex))
         (y3 (vertex-y-coord s-vertex))
         (vlist (edge-vertex-list s-edge))
         (s-vertex1 (eval (first vlist)))
         (s-vertex2 (eval (second vlist)))
         (x1 (vertex-x-coord s-vertex1))
         (y1 (vertex-y-coord s-vertex1))
         (x2 (vertex-x-coord s-vertex2))
         (y2 (vertex-y-coord s-vertex2))
         (lequation (line-equation x1 y1 x2 y2))
         (a (first lequation))
         (b (second lequation))
         (c (third lequation))
         (distance1 (line-length x1 y1 x3 y3))
         (distance2 (line-length x2 y2 x3 y3))
         (distance3 (/ (abs (+ (* a x3) (* b y3) c))
                       (sqrt (+ (* a a) (* b b)))))
         (vector1 (list (- x3 x1) (- y3 y1)))
         (vector2 (list (- x2 x1) (- y2 y1))))
    (if (null (vector-project vector1 vector2))
        (max distance1 distance2)
        (max distance1 distance2 distance3))))


(defun distance-edge-to-edge-min (edge1 edge2)
  (let* ((vertex-list1 (edge-vertex-list (eval edge1)))
         (vertex-list2 (edge-vertex-list (eval edge2)))
         (vertex11 (first vertex-list1))
         (vertex12 (second vertex-list1))
         (vertex21 (first vertex-list2))
         (vertex22 (second vertex-list2))
         (distance1 (distance-vertex-to-edge-min vertex11 edge2))
         (distance2 (distance-vertex-to-edge-min vertex12 edge2))
         (distance3 (distance-vertex-to-edge-min vertex21 edge1))
         (distance4 (distance-vertex-to-edge-min vertex22 edge1)))
    (min distance1 distance2 distance3 distance4)))
```

```
(defun distance-edge-to-edge-max (edge1 edge2)
  (let* ((vertex-list1 (edge-vertex-list (eval edge1)))
         (vertex-list2 (edge-vertex-list (eval edge2)))
         (vertex11 (first vertex-list1))
         (vertex12 (second vertex-list1))
         (vertex21 (first vertex-list2))
         (vertex22 (second vertex-list2))
         (distance1 (distance-vertex-to-edge-max vertex11 edge2))
         (distance2 (distance-vertex-to-edge-max vertex12 edge2))
         (distance3 (distance-vertex-to-edge-max vertex21 edge1))
         (distance4 (distance-vertex-to-edge-max vertex22 edge1)))
    (max distance1 distance2 distance3 distance4)))


(defun distance-window-to-window-lower (window1 window2)
  (let* ((s-window1 (eval window1))
         (s-window2 (eval window2))
         (window1-vertex (if (vertex-p s-window1) t nil))
         (window2-vertex (if (vertex-p s-window2) t nil))
         (window-distance nil))
    (setf window-distance (cond ((and window1-vertex window2-vertex)
                                 (distance-vertex-to-vertex window1 window2))
                                ((and (null window1-vertex)
                                      (null window2-vertex))
                                 (distance-edge-to-edge-min window1 window2))
                                ((and window1-vertex (null window2-vertex))
                                 (distance-vertex-to-edge-min window1 window2))
                                ((and (null window1-vertex) window2-vertex)
                                 (distance-vertex-to-edge-min
                                   window2 window1))))))


(defun distance-window-to-window-upper (window1 window2)
  (let* ((s-window1 (eval window1))
         (s-window2 (eval window2))
         (window1-vertex (if (vertex-p s-window1) t nil))
         (window2-vertex (if (vertex-p s-window2) t nil))
         (window-distance nil))
    (setf window-distance (cond ((and window1-vertex window2-vertex)
                                 (distance-vertex-to-vertex window1 window2))
                                ((and (null window1-vertex)
                                      (null window2-vertex))
                                 (distance-edge-to-edge-max window1 window2))
                                ((and window1-vertex (null window2-vertex))
                                 (distance-vertex-to-edge-max window1 window2))
                                ((and (null window1-vertex) window2-vertex)
                                 (distance-vertex-to-edge-max
                                   window2 window1))))))


;*****************************************************************************
;
; Geometric Model: Heading Functions
;
;*****************************************************************************


(defun quadrant (x1 y1 x2 y2)
  (cond ((and (> x2 x1) (> y2 y1)) 'ne)
        ((and (< x2 x1) (> y2 y1)) 'nw)
```

```
        ((and (< x2 x1) (< y2 y1)) 'sw)
        ((and (> x2 x1) (< y2 y1)) 'se)
        ((and (= x2 x1) (> y2 y1)) 'n)
        ((and (< x2 x1) (= y2 y1)) 'w)
        ((and (= x2 x1) (< y2 y1)) 's)
        ((and (> x2 x1) (= y2 y1)) 'e)))


(defun heading-quadrant (heading)
  (if (= (mod heading 360.0) 0.0) 'n
      (if (= (mod heading 360.0) 90.0) 'e
          (if (= (mod heading 360.0) 180.0) 's
              (if (= (mod heading 360.0) 270.0) 'w
                  (if (and (> (mod heading 360.0) 0.0)
                           (< (mod heading 360.0) 90.0)) 'ne
                      (if (and (> (mod heading 360.0) 90.0)
                               (< (mod heading 360.0) 180.0)) 'se
                          (if (and (> (mod heading 360.0) 180.0)
                                   (< (mod heading 360.0) 270.0)) 'sw
                              (if (and (> (mod heading 360.0) 270.0)
                                       (< (mod heading 360.0) 360.0))
                                  'nw))))))))


(defun quad1-heading (x1 y1 x2 y2)
  (if (point-equal-p x1 y1 x2 y2) nil
      (- 90.0 (radians-to-degrees (atan (abs (- y1 y2))
              (abs (- x1 x2)))))))


(defun heading (x1 y1 x2 y2)
  (let ((quad (quadrant x1 y1 x2 y2))
        (q1heading (quad1-heading x1 y1 x2 y2)))
    (cond ((equal quad 'ne) q1heading)
          ((equal quad 'nw) (- 360.0 q1heading))
          ((equal quad 'sw) (+ 180.0 q1heading))
          ((equal quad 'se) (- 180.0 q1heading))
          ((equal quad 'n) 0.0)
          ((equal quad 'w) 270.0)
          ((equal quad 's) 180.0)
          ((equal quad 'e) 90.0))))


(defun reverse-heading (heading)
  (mod (+ heading 180.0) 360.0))


(defun normalize-heading (heading)
  (mod heading 360.0))


(defun order-headings (heading1 heading2)
  (let* ((heading-1 (normalize-heading heading1))
         (heading-2 (normalize-heading heading2))
         (heading-quadrant1 (heading-quadrant heading-1))
         (heading-quadrant2 (heading-quadrant heading-2))
         (greater-heading (max heading-1 heading-2))
         (lesser-heading (if (= greater-heading heading-1) heading-2 heading-1))
         (tolerance 0.01))
    (if (and (same-quadrant-p heading-1 heading-2)
```

```
     (= heading-1 lesser-heading))
(list heading-1 heading-2)
(if (and (same-quadrant-p heading-1 heading-2)
         (= heading-1 greater-heading))
    (list heading-2 heading-1)
    (if (or (and (equal heading-quadrant1 'n)
                 (equal heading-quadrant2 'ne))
            (and (equal heading-quadrant1 'n)
                 (equal heading-quadrant2 'e))
            (and (equal heading-quadrant1 'n)
                 (equal heading-quadrant2 'se))
            (and (equal heading-quadrant1 'sw)
                 (equal heading-quadrant2 'n))
            (and (equal heading-quadrant1 'w)
                 (equal heading-quadrant2 'n))
            (and (equal heading-quadrant1 'nw)
                 (equal heading-quadrant2 'n))
            (and (equal heading-quadrant1 'e)
                 (equal heading-quadrant2 'se))
            (and (equal heading-quadrant1 'e)
                 (equal heading-quadrant2 's))
            (and (equal heading-quadrant1 'e)
                 (equal heading-quadrant2 'sw))
            (and (equal heading-quadrant1 'nw)
                 (equal heading-quadrant2 'e))
            (and (equal heading-quadrant1 'ne)
                 (equal heading-quadrant2 'e))
            (and (equal heading-quadrant1 's)
                 (equal heading-quadrant2 'sw))
            (and (equal heading-quadrant1 's)
                 (equal heading-quadrant2 'w))
            (and (equal heading-quadrant1 's)
                 (equal heading-quadrant2 'nw))
            (and (equal heading-quadrant1 'ne)
                 (equal heading-quadrant2 's))
            (and (equal heading-quadrant1 'se)
                 (equal heading-quadrant2 's))
            (and (equal heading-quadrant1 'w)
                 (equal heading-quadrant2 'nw))
            (and (equal heading-quadrant1 'w)
                 (equal heading-quadrant2 'ne))
            (and (equal heading-quadrant1 'se)
                 (equal heading-quadrant2 'w))
            (and (equal heading-quadrant1 'sw)
                 (equal heading-quadrant2 'w))
            (and (equal heading-quadrant1 'ne)
                 (equal heading-quadrant2 'se))
            (and (equal heading-quadrant1 'nw)
                 (equal heading-quadrant2 'ne))
            (and (equal heading-quadrant1 'se)
                 (equal heading-quadrant2 'sw))
            (and (equal heading-quadrant1 'sw)
                 (equal heading-quadrant2 'nw)))
        (list heading-1 heading-2)
        (if (or (and (equal heading-quadrant2 'n)
                     (equal heading-quadrant1 'ne))
                (and (equal heading-quadrant2 'n)
                     (equal heading-quadrant1 'e))
                (and (equal heading-quadrant2 'n)
                     (equal heading-quadrant1 'se))
```

```
                    (and (equal heading-quadrant2 'sw)
                         (equal heading-quadrant1 'n))
                    (and (equal heading-quadrant2 'w)
                         (equal heading-quadrant1 'n))
                    (and (equal heading-quadrant2 'nw)
                         (equal heading-quadrant1 'n))
                    (and (equal heading-quadrant2 'e)
                         (equal heading-quadrant1 'se))
                    (and (equal heading-quadrant2 'e)
                         (equal heading-quadrant1 's))
                    (and (equal heading-quadrant2 'e)
                         (equal heading-quadrant1 'sw))
                    (and (equal heading-quadrant2 'nw)
                         (equal heading-quadrant1 'e))
                    (and (equal heading-quadrant2 'ne)
                         (equal heading-quadrant1 'e))
                    (and (equal heading-quadrant2 's)
                         (equal heading-quadrant1 'sw))
                    (and (equal heading-quadrant2 's)
                         (equal heading-quadrant1 'w))
                    (and (equal heading-quadrant2 's)
                         (equal heading-quadrant1 'nw))
                    (and (equal heading-quadrant2 'ne)
                         (equal heading-quadrant1 's))
                    (and (equal heading-quadrant2 'se)
                         (equal heading-quadrant1 's))
                    (and (equal heading-quadrant2 'w)
                         (equal heading-quadrant1 'nw))
                    (and (equal heading-quadrant2 'w)
                         (equal heading-quadrant1 'ne))
                    (and (equal heading-quadrant2 'se)
                         (equal heading-quadrant1 'w))
                    (and (equal heading-quadrant2 'sw)
                         (equal heading-quadrant1 'w))
                    (and (equal heading-quadrant2 'ne)
                         (equal heading-quadrant1 'se))
                    (and (equal heading-quadrant2 'nw)
                         (equal heading-quadrant1 'ne))
                    (and (equal heading-quadrant2 'se)
                         (equal heading-quadrant1 'sw))
                    (and (equal heading-quadrant2 'sw)
                         (equal heading-quadrant1 'nw)))
                (list heading-2 heading-1)
                (if (< heading-2 (mod (+ heading-1 180.0) 360.0))
                    (list heading-1 heading-2)
                    (if (> heading-2 (mod (+ heading-1 180.0) 360.0))
                        (list heading-2 heading-1)
                        (if (or (equal-within-tolerance
                                  heading-2 (mod (+ heading-1 180.0)
                                    360.0) tolerance)
                                (equal-within-tolerance
                                  heading-1 (mod (+ heading-2 180.0)
                                    360.0) tolerance))
                            nil)))))))))


(defun interval-order-headings (heading-list-1 heading-list-2)
  (let* ((heading-1 (normalize-heading (first heading-list-1)))
         (heading-designation-1 (second heading-list-1))
         (heading-2 (normalize-heading (first heading-list-2)))
```

```
          (heading-designation-2 (second heading-list-2))
          (ordered-headings (order-headings heading-1 heading-2))
          (ordered-heading-list-1
            (if ordered-headings
                (if (= (first ordered-headings) heading-1)
                    (list heading-1 heading-designation-1)
                    (list heading-2 heading-designation-2))))
          (ordered-heading-list-2
            (if ordered-headings
                (if (= (second ordered-headings) heading-1)
                    (list heading-1 heading-designation-1)
                    (list heading-2 heading-designation-2)))))
      (if ordered-headings
          (list ordered-heading-list-1 ordered-heading-list-2))))


(defun heading-inclination-angle (heading slope orientation)
  (radians-to-degrees
    (atan (* (- (cos (degrees-to-radians (- orientation heading))))
      (tan (degrees-to-radians slope))))))


(defun heading-point (x1 y1 vheading)
  (let* ((nw-x (vertex-x-coord (eval 'v-nw)))
         (nw-y (vertex-y-coord (eval 'v-nw)))
         (ne-x (vertex-x-coord (eval 'v-ne)))
         (ne-y (vertex-y-coord (eval 'v-ne)))
         (sw-x (vertex-x-coord (eval 'v-sw)))
         (sw-y (vertex-y-coord (eval 'v-sw)))
         (se-x (vertex-x-coord (eval 'v-se)))
         (se-y (vertex-y-coord (eval 'v-se)))
         (ymin 0.0)
         (ymax 559.0)
         (tan-theta (if (or (= vheading 0.0) (= vheading 90.0)
                            (= vheading 180.0) (= vheading 270.0)) nil
                        (tan (degrees-to-radians (- 90.0 vheading)))))
         (x2
           (if (and (> vheading 270.0) (< vheading 360.0))
               (+ x1 (/ (- ymax y1) tan-theta))
               (if (and (> vheading 90.0) (< vheading 180.0))
                   (+ x1 (/ (- ymin y1) tan-theta))
                   (if (and (> vheading 0.0) (< vheading 90.0))
                       (+ x1 (/ (- ymax y1) tan-theta))
                       (if (and (> vheading 180.0) (< vheading 270.0))
                           (+ x1 (/ (- ymin y1) tan-theta)))))))
         (intersection-point
           (if (= vheading 0.0)
               (list x1 (vertex-y-coord
                          (eval (first (edge-vertex-list (eval 'e-n))))))
               (if (= vheading 180.0)
                   (list x1 (vertex-y-coord
                              (eval (first (edge-vertex-list (eval 'e-s))))))
                   (if (= vheading 90.0)
                       (list (vertex-x-coord
                               (eval (first (edge-vertex-list
                                      (eval 'e-e))))) y1)
                       (if (= vheading 270.0)
                           (list (vertex-x-coord
                                   (eval (first (edge-vertex-list
                                         (eval 'e-w))))) y1)
```

```
                                (if (and (> vheading 270.0) (< vheading 360.0))
                                    (list x2 ymax)
                                    (if (and (> vheading 90.0) (< vheading 180.0))
                                        (list x2 ymin)
                                        (if (and (> vheading 0.0) (< vheading 90.0))
                                            (list x2 ymax)
                                            (if (and (> vheading 180.0)
                                                     (< vheading 270.0))
                                                (list x2 ymin)))))))))
          (intersection-point-x (first intersection-point))
          (intersection-point-y (second intersection-point))
          (intercept-north (intersect x1 y1 intersection-point-x
                                              intersection-point-y
                                              nw-x nw-y ne-x ne-y))
          (intercept-south (intersect x1 y1 intersection-point-x
                                              intersection-point-y
                                              sw-x sw-y se-x se-y))
          (intercept-east (intersect x1 y1 intersection-point-x
                                             intersection-point-y
                                             ne-x ne-y se-x se-y))
          (intercept-west (intersect x1 y1 intersection-point-x
                                             intersection-point-y
                                             nw-x nw-y sw-x sw-y))
          (heading-point
            (first (remove-if 'null
                   (list intercept-north intercept-south
                         intercept-east intercept-west))))) heading-point))


(defun heading-equal-p (heading1 heading2)
  (let ((tolerance 0.01))
    (if (equal-within-tolerance heading1 heading2 tolerance) t nil)))


(defun heading-range-p (heading heading1 heading2)
  (let* ((ordered-headings (order-headings heading1 heading2)))
    (if ordered-headings
        (let* ((heading-1 (first ordered-headings))
               (heading-2 (second ordered-headings))
               (tolerance 0.01))
          (if (or (equal-within-tolerance heading heading-1 tolerance)
                  (equal-within-tolerance heading heading-2 tolerance)
                  (and (> heading-1 heading-2)
                       (or (not (> heading heading-2))
                           (not (< heading heading-1))))
                  (and (> heading heading-1)
                       (< heading heading-2))) t)))))


(defun interval-heading-range-p (heading-list heading-list-1 heading-list-2)
  (let* ((heading-1 (first heading-list-1))
         (heading-2 (first heading-list-2))
         (ordered-headings (order-headings heading-1 heading-2)))
    (if ordered-headings
        (let* ((heading (first heading-list))
               (ordered-heading-1 (first ordered-headings))
               (ordered-heading-2 (second ordered-headings))
               (tolerance 0.01)
               (limit-1
                 (equal-within-tolerance heading ordered-heading-1 tolerance))
```

```
                    (limit-2
                      (equal-within-tolerance heading heading-2 tolerance))
                    (heading-designation (second heading-list))
                    (heading-designation-1 (second heading-list-1))
                    (heading-designation-2 (second heading-list-2)))
               (if (or limit-1 limit-2)
                   (if (or (and limit-1
                                (equal heading-designation 'CL)
                                (equal heading-designation-1 'CL))
                           (and limit-2
                                (equal heading-designation 'CL)
                                (equal heading-designation-2 'CL))) t)
                   (if (or (and (> ordered-heading-1 ordered-heading-2)
                                (or (not (> heading ordered-heading-2))
                                    (not (< heading ordered-heading-1))))
                           (and (> heading ordered-heading-1)
                                (< heading ordered-heading-2))) t))))))


(defun heading-range-intersection (heading-range-1 heading-range-2)
  (let* ((heading-range-1-full
           (if (= (length heading-range-1) 2) t nil))
         (heading-range-1-partial
           (if (= (length heading-range-1) 1) t nil))
         (heading-range-2-full
           (if (= (length heading-range-2) 2) t nil))
         (heading-range-2-partial
           (if (= (length heading-range-2) 1) t nil))
         (ordered-heading-range-1-full
           (if heading-range-1-full
               (order-headings
                 (first heading-range-1)
                 (second heading-range-1))))
         (ordered-heading-range-2-full
           (if heading-range-2-full
               (order-headings
                 (first heading-range-2)
                 (second heading-range-2))))
         (tolerance 0.01))
    (if
      (and heading-range-1-partial
           heading-range-2-partial
           (equal-within-tolerance
             (first heading-range-1) (first heading-range-2) tolerance))
      heading-range-1
      (if
        (and heading-range-1-partial
             heading-range-2-full
             (heading-range-p (first heading-range-1)
                              (first ordered-heading-range-2-full)
                              (second ordered-heading-range-2-full)))
        heading-range-1
        (if
          (and heading-range-1-full
               heading-range-2-partial
               (heading-range-p (first heading-range 2)
                                (first ordered-heading-range-1-full)
                                (second ordered-heading-range-1-full)))
          heading-range-2
          (if
```

```
(and ordered-heading-range-1-full
     ordered-heading-range-2-full)
(if
  (and (heading-range-p (first ordered-heading-range-2-full)
                        (first ordered-heading-range-1-full)
                        (second ordered-heading-range-1-full))
       (heading-range-p (second ordered-heading-range-2-full)
                        (first ordered-heading-range-1-full)
                        (second ordered-heading-range-1-full)))
  (order-headings (first ordered-heading-range-2-full)
                  (second ordered-heading-range-2-full))
  (if
    (and (heading-range-p (first ordered-heading-range-1-full)
                          (first ordered-heading-range-2-full)
                          (second ordered-heading-range-2-full))
         (heading-range-p (second ordered-heading-range-1-full)
                          (first ordered-heading-range-2-full)
                          (second ordered-heading-range-2-full)))
    (order-headings (first ordered-heading-range-1-full)
                    (second ordered-heading-range-1-full))
    (if
      (or
        (and (heading-range-p (first ordered-heading-range-2-full)
                              (first ordered-heading-range-1-full)
                              (second ordered-heading-range-1-full))
             (null (heading-range-p
                     (second ordered-heading-range-2-full)
                     (first ordered-heading-range-1-full)
                     (second ordered-heading-range-1-full))))
        (and (heading-range-p (second ordered-heading-range-1-full)
                              (first ordered-heading-range-2-full)
                              (second ordered-heading-range-2-full))
             (null (heading-range-p
                     (first ordered-heading-range-1-full)
                     (first ordered-heading-range-2-full)
                     (second ordered-heading-range-2-full)))))
      (if
        (equal-within-tolerance
          (first ordered-heading-range-2-full)
          (second ordered-heading-range-1-full)
          tolerance)
        (list (first ordered-heading-range-2-full))
        (list (first ordered-heading-range-2-full)
              (second ordered-heading-range-1-full)))
      (if
        (or (and (heading-range-p
                   (second ordered-heading-range-2-full)
                   (first ordered-heading-range-1-full)
                   (second ordered-heading-range-1-full))
                 (null (heading-range-p
                         (first ordered-heading-range-2-full)
                         (first ordered-heading-range-1-full)
                         (second ordered-heading-range-1-full))))
            (and (heading-range-p
                   (first ordered-heading-range-1-full)
                   (first ordered-heading-range-2-full)
                   (second ordered-heading-range-2-full))
                 (null (heading-range-p
                         (second ordered-heading-range-1-full)
                         (first ordered-heading-range-2-full)
```

```
                                    (second ordered-heading-range-2-full)))))
                          (if (equal-within-tolerance
                                (second ordered-heading-range-2-full)
                                (first ordered-heading-range-1-full)
                                tolerance)
                              (list (second ordered-heading-range-2-full))
                              (list (first ordered-heading-range-1-full)
                                    (second ordered-heading-range-2-full)))))))))))))


(defun interval-heading-range-intersection (heading-range-1 heading-range-2)
  (let* ((heading-range-1-full
           (if (= (length heading-range-1) 2) t nil))
         (heading-range-1-partial
           (if (= (length heading-range-1) 1) t nil))
         (heading-range-2-full
           (if (= (length heading-range-2) 2) t nil))
         (heading-range-2-partial
           (if (= (length heading-range-2) 1) t nil))
         (tolerance 0.01))
    (if (and heading-range-1-full heading-range-2-full)
        (let* ((ordered-heading-range-1
                 (interval-order-headings
                   (first heading-range-1) (second heading-range-1)))
               (ordered-heading-range-2
                 (interval-order-headings
                   (first heading-range-2) (second heading-range-2)))
               (heading-list-11 (first ordered-heading-range-1))
               (heading-11 (first heading-list-11))
               (heading-designation-11 (second heading-list-11))
               (heading-list-12 (second ordered-heading-range-1 ))
               (heading-12 (first heading-list-12))
               (heading-designation-12 (second heading-list-12))
               (heading-list-21 (first ordered-heading-range-2))
               (heading-21 (first heading-list-21))
               (heading-designation-21 (second heading-list-21))
               (heading-list-22 (second ordered-heading-range-2))
               (heading-22 (first heading-list-22))
               (heading-designation-22 (second heading-list-22))
               (partial-heading-range-intersection
                 (if (= (length (heading-range-intersection
                                  (list (first (first heading-range-1))
                                        (first (second heading-range-1)))
                                  (list (first (first heading-range-2))
                                        (first (second heading-range-2)))))) 1)
                                  t nil))
               (heading-limit-21-11
                 (if (equal-within-tolerance heading-11 heading-21 tolerance)
                     t nil))
               (heading-limit-21-12
                 (if (equal-within-tolerance heading-12 heading-21 tolerance)
                     t nil))
               (heading-limit-21
                 (if (or heading-limit-21-11 heading-limit-21-12) t nil))
               (heading-limit-closed-21
                 (if (or (and heading-limit-21-11
                              (equal heading-designation-11 'CL)
                              (equal heading-designation-21 'CL))
                         (and heading-limit-21-12
                              (equal heading-designation-12 'CL)
```

```
                    (equal heading-designation-21 'CL))) t nil))
      (heading-limit-22-11
        (if (equal-within-tolerance heading-11 heading-22 tolerance)
            t nil))
      (heading-limit-22-12
        (if (equal-within-tolerance heading-12 heading-22 tolerance)
            t nil))
      (heading-limit-22
        (if (or heading-limit-22-11 heading-limit-22-12) t nil))
      (heading-limit-closed-22
        (if (or (and heading-limit-22-11
                     (equal heading-designation-11 'CL)
                     (equal heading-designation-22 'CL))
                (and heading-limit-22-12
                     (equal heading-designation-12 'CL)
                     (equal heading-designation-22 'CL))) t nil))
      (heading-limits
        (remove-if 'null (list heading-limit-21 heading-limit-22))))
(if (null heading-limits)
    (let ()
      (if (and (interval-heading-range-p heading-list-21
                                         heading-list-11
                                         heading-list-12)
               (interval-heading-range-p heading-list-22
                                         heading-list-11
                                         heading-list-12))
          (interval-order-headings heading-list-21 heading-list-22)
          (if (and (interval-heading-range-p heading-list-11
                                             heading-list-21
                                             heading-list-22)
                   (interval-heading-range-p heading-list-12
                                             heading-list-21
                                             heading-list-22))
              (interval-order-headings heading-list-11
                                       heading-list-12)
              (if (or (and (interval-heading-range-p heading-list-11
                                                     heading-list-21
                                                     heading-list-22)
                           (null (interval-heading-range-p
                                  heading-list-12
                                  heading-list-21
                                  heading-list-22)))
                      (and (interval-heading-range-p heading-list-22
                                                     heading-list-11
                                                     heading-list-12)
                           (null (interval-heading-range-p
                                  heading-list-21
                                  heading-list-11
                                  heading-list-12))))
                  (interval-order-headings heading-list-11
                                           heading-list-22)
                  (if (or (and (interval-heading-range-p
                                heading-list-21
                                heading-list-11
                                heading-list-12)
                               (null (interval-heading-range-p
                                heading-list-22
                                heading-list-11
                                heading-list-12)))
                          (and (interval-heading-range-p
```

```
                                        heading-list-12
                                        heading-list-21
                                        heading-list-22)
                                      (null (interval-heading-range-p
                                        heading-list-11
                                        heading-list-21
                                        heading-list-22))))
                              (interval-order-headings
                                heading-list-21 heading-list-12))))))
                (if partial-heading-range-intersection
                    (let ()
                      (if heading-limit-closed-21
                          (list (list heading-21 'CL))
                          (if heading-limit-closed-22
                              (list (list heading-22 'CL)))))
                    (if (= (length heading-limits) 2)
                        (let ()
                          (if (and heading-limit-closed-21 heading-limit-closed-22)
                              (interval-order-headings
                                (list heading-21 'CL) (list heading-22 'CL))
                              (if (and (null heading-limit-closed-21)
                                       heading-limit-closed-22)
                                  (interval-order-headings
                                    (list heading-21 'OP) (list heading-22 'CL))
                                  (if (and heading-limit-closed-21
                                           (null heading-limit-closed-22))
                                      (interval-order-headings
                                        (list heading-21 'CL) (list heading-22 'OP))
                                      (if (and (null heading-limit-closed-21)
                                               (null heading-limit-closed-22))
                                          (interval-order-headings
                                            (list heading-21 'OP)
                                            (list heading-22 'OP)))))))
                        (if (= (length heading-limits) 1)
                            (let ()
                              (if heading-limit-21
                                  (let ()
                                    (if (interval-heading-range-p heading-list-22
                                                                  heading-list-11
                                                                  heading-list-12)
                                        (if heading-limit-closed-21
                                            (interval-order-headings
                                              (list heading-21 'CL) heading-list-22)
                                            (interval-order-headings
                                              (list heading-21 'OP) heading-list-22))
                                        (if (null (interval-heading-range-p
                                                    heading-list-22
                                                    heading-list-11
                                                    heading-list-12))
                                            (if heading-limit-closed-21
                                                (interval-order-headings
                                                  (list heading-21 'CL)
                                                  heading-list-12)
                                                (interval-order-headings
                                                  (list heading-21 'OP)
                                                  heading-list-12))
                                            (if (interval-heading-range-p
                                                  heading-list-12
                                                  heading-list-21
                                                  heading-list-22)
```

```
                    (if heading-limit-closed-21
                        (interval-order-headings
                          (list heading-21 'CL)
                          heading-list-12)
                        (interval-order-headings
                          (list heading-21 'OP)
                          heading-list-12))
                    (if (null (interval-heading-range-p
                                heading-list-12
                                heading-list-21
                                heading-list-22))
                        (if heading-limit-closed-21
                            (interval-order-headings
                              (list heading-21 'CL)
                              heading-list-22)
                            (interval-order-headings
                              (list heading-21 'OP)
                              heading-list-22)))))))
(if heading-limit-22
    (let ()
      (if (interval-heading-range-p heading-list-21
                                    heading-list-11
                                    heading-list-12)
          (if heading-limit-closed-22
              (interval-order-headings
                (list heading-22 'CL)
                heading-list-21)
              (interval-order-headings
                (list heading-22 'OP)
                heading-list-21))
          (if (null (interval-heading-range-p
                      heading-list-21
                      heading-list-11
                      heading-list-12))
              (if heading-limit-closed-22
                  (interval-order-headings
                    (list heading-22 'CL)
                    heading-list-11)
                  (interval-order-headings
                    (list heading-22 'OP)
                    heading-list-11))
              (if (interval-heading-range-p
                    heading-list-11
                    heading-list-21
                    heading-list-22)
                  (if heading-limit-closed-22
                      (interval-order-headings
                        (list heading-22 'CL)
                        heading-list-11)
                      (interval-order-headings
                        (list heading-22 'OP)
                        heading-list-11))
                  (if
                    (null (interval-heading-range-p
                            heading-list-11
                            heading-list-21
                            heading-list-22))
                    (if heading-limit-closed-22
                        (interval-order-headings
                          (list heading-22 'CL)
```

```
                                                    heading-list-21)
                                                  (interval-order-headings
                                                    (list heading-22 'OP)
                                                  heading-list-21))))))))))))))))
          (if (and heading-range-1-partial heading-range-2-partial)
              (let* ((heading-list-11 (first heading-range-1))
                     (heading-list-21 (first heading-range-2))
                     (heading-11 (first heading-list-11))
                     (heading-21 (first heading-list-21))
                     (heading-designation-11 (second heading-list-11))
                     (heading-designation-21 (second heading-list-21)))
                (if (and (equal-within-tolerance heading-11 heading-21 tolerance)
                         (equal heading-designation-11 'CL)
                         (equal heading-designation-21 'CL)) heading-range-1))
              (if (and heading-range-1-full heading-range-2-partial)
                  (let ((heading-list-21 (first heading-range-2))
                        (heading-list-11 (first heading-range-1))
                        (heading-list-12 (second heading-range-1)))
                    (if (interval-heading-range-p heading-list-21
                                                  heading-list-11
                                                  heading-list-12)
                        heading-range-2))
                  (if (and heading-range-1-partial heading-range-2-full)
                      (let ((heading-list-11 (first heading-range-1))
                            (heading-list-21 (first heading-range-2))
                            (heading-list-22 (second heading-range-2)))
                        (if (interval-heading-range-p heading-list-11
                                                      heading-list-21
                                                      heading-list-22)
                            heading-range-1))))))))


(defun same-quadrant-p (heading1 heading2)
  (if (or (and (>= heading1 0.0)
               (<= heading1 90.0)
               (>= heading2 0.0)
               (<= heading2 90.0))
          (and (>= heading1 90.0)
               (<= heading1 180.0)
               (>= heading2 90.0)
               (<= heading2 180.0))
          (and (>= heading1 180.0)
               (<= heading1 270.0)
               (>= heading2 180.0)
               (<= heading2 270.0))
          (and (>= heading1 270.0)
               (< heading1 360.0)
               (>= heading2 270.0)
               (< heading2 360.0))) t nil))


(defun total-heading-range-p (heading heading-11 heading-12
                              heading-21 heading-22)
  (let* ((heading-12 (if (= heading-12 0.0) 360.0 heading-12))
         (heading-21 (if (= heading-21 0.0) 360.0 heading-21)))
    (cond ((and (> heading-11 heading-12)
                (or (not (and (> heading heading-12)
                              (< heading heading-11)))
                    (and (< heading heading-21)
                         (> heading heading-22)))) t)
```

```
          ((and (> heading-22 heading-21)
                (or (not (and (> heading heading-21)
                              (< heading heading-22)))
                         (and (<= heading heading-12)
                              (>= heading heading-11)))) t)
          ((or (and (> heading heading-11)
                    (< heading heading-12))
               (and (<= heading heading-21)
                    (>= heading heading-22))) t)
          (t nil))))


(defun braking-heading-p (heading region)
  (if (or (isotropic-region-p region)
          (obstacle-region-p region)) nil
      (let ((heading-incl-angle
              (heading-inclination-angle
                heading (region-slope (eval region))
                (region-orientation (eval region)))))
        (if (and (minusp heading-incl-angle)
                 (>= (abs heading-incl-angle) *critical-coasting-angle*))
            t nil))))


(defun stability-heading-p (heading region)
  (if (member heading (region-stability-constraints (eval region))) t nil))


(defun braking-headings (critical-coasting-angle slope orientation)
  (let* ((adjusted-orientation (if (= orientation 0.0) 360.0 orientation))
         (braking-heading1
           (- adjusted-orientation
             (radians-to-degrees
               (acos (/ (tan (degrees-to-radians critical-coasting-angle))
                 (tan (degrees-to-radians slope)))))))
         (braking-heading2
           (- (mod (+ 180.0 adjusted-orientation) 360.0)
             (radians-to-degrees
               (acos (- (/ (tan (degrees-to-radians critical-coasting-angle))
                 (tan (degrees-to-radians slope))))))))))
    (order-headings (make-significant-figures
                      (normalize-heading braking-heading1) 3)
                    (make-significant-figures
                      (normalize-heading braking-heading2) 3))))


(defun edge-slope (edge)
  (let* ((region-list (edge-adjacency-list (eval edge)))
         (region (first region-list))
         (slope (region-slope (eval region)))
         (orientation (region-orientation (eval region)))
         (vertex-list (edge-vertex-list (eval edge)))
         (vertex1 (first vertex-list))
         (vertex2 (second vertex-list))
         (v1-x (vertex-x-coord (eval vertex1)))
         (v1-y (vertex-y-coord (eval vertex1)))
         (v2-x (vertex-x-coord (eval vertex2)))
         (v2-y (vertex-y-coord (eval vertex2)))
         (edge-heading (heading v1-x v1-y v2-x v2-y)))
    (make-significant-figures
```

```
      (abs (heading-inclination-angle edge-heading slope orientation)) 3)))


(defun braking-entry-angle (edge-slope)
  (radians-to-degrees
    (asin (/ (tan (degrees-to-radians edge-slope)) *motion-resistance-lower*))))


(defun stability-headings (stability-offset region)
  (if (anisotropic-partially-safe-region-p region)
      (let* ((gradient-list (gradient-heading region))
             (gradient-down (first gradient-list))
             (gradient-up (second gradient-list))
             (down-heading-1 (- gradient-down stability-offset))
             (down-heading-2 (+ gradient-down stability-offset))
             (up-heading-1 (+ gradient-up stability-offset))
             (up-heading-2 (- gradient-up stability-offset))
             (heading-1 (if (minusp down-heading-1) (+ down-heading-1 360.0)
               down-heading-1))
             (heading-2 (mod down-heading-2 360.0))
             (heading-3 (if (minusp up-heading-2) (+ up-heading-2 360.0)
               up-heading-2))
             (heading-4 (mod up-heading-1 360.0)))
        (list heading-1 heading-2 heading-3 heading-4))))


(defun gradient-heading (region)
  (let* ((rorientation (region-orientation (eval region))))
    (if (null rorientation) nil
        (list rorientation (mod (+ rorientation 180.0) 360.0)))))


(defun contour-heading (region)
  (let* ((rorientation (region-orientation (eval region))))
    (if (null rorientation) nil
        (list (+ 90.0 rorientation) (mod (+ rorientation 270.0) 360.0)))))


;*******************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-SEARCH-UTILITIES-I
;
; Functions: PERMISSIBLE-HEADING-RANGE-FROM-EDGE-TO-EDGE-NON-INCIDENT
;             (frontier-window1 frontier-window2)
;            PERMISSIBLE-HEADING-RANGE-FROM-EDGE-TO-EDGE-INCIDENT
;             (frontier-window1 frontier-window2)
;            PERMISSIBLE-HEADING-RANGE-FROM-EDGE-TO-EDGE-OBSCURE
;             (frontier-window1 frontier-window1-approach-region
;              frontier-window2 frontier-window2-approach-region)
;            PERMISSIBLE-HEADING-RANGE-FROM-EDGE-TO-EDGE
;             (frontier-window1 frontier-window2)
;            PERMISSIBLE-HEADING-RANGE-FROM-EDGE-TO-VERTEX
;             (frontier-window1 frontier-window2)
;            PERMISSIBLE-HEADING-RANGE-FROM-VERTEX-TO-EDGE
;             (frontier-window1 frontier-window2)
;            PERMISSIBLE-HEADING-RANGE-FROM-VERTEX-TO-VERTEX
;             (frontier-window1 frontier-window2)
;
;            PERMISSIBLE-HEADINGS-CRITICAL-STABILITY (frontier-region)
;            IMPERMISSIBLE-HEADINGS-CRITICAL-INSTABILITY (frontier-region)
;            PERMISSIBLE-HEADINGS-CRITICAL-BRAKING (frontier-region)
;
;            PERMISSIBLE-HEADINGS-GEOMETRIC
;             (frontier-window1 frontier-window2);
;            PERMISSIBLE-HEADINGS-STABILITY
;             (frontier-window1 frontier-window2)
;            PERMISSIBLE-HEADINGS-BRAKING
;             (frontier-window1 frontier-window2)
;            PERMISSIBLE-HEADINGS-NON-BRAKING
;             (frontier-window1 frontier-window2)
;
;            PERMISSIBLE-HEADINGS-INTERSECTION
;             (PERMISSIBLE-headings-1 PERMISSIBLE-headings-2)
;
;*****************************************************************************


;*****************************************************************************
;
; Path Planning Model: Construction of Permissible Heading Ranges
;                      (Geometric Constraints)
;
;*****************************************************************************


(defun permissible-heading-range-from-edge-to-edge-non-incident
       (frontier-window1 frontier-window2)
  (let* ((frontier-window1-vertexlist
            (edge-vertex-list (eval frontier-window1)))
         (fw1-v1 (first frontier-window1-vertexlist))
         (fw1-v2 (second frontier-window1-vertexlist))
         (fw1-v1-x (vertex-x-coord (eval fw1-v1)))
         (fw1-v1-y (vertex-y-coord (eval fw1-v1)))
         (fw1-v2-x (vertex-x-coord (eval fw1-v2)))
         (fw1-v2-y (vertex-y-coord (eval fw1-v2)))
         (frontier-window2-vertexlist
           (edge-vertex-list (eval frontier-window2)))
         (fw2-v1 (first frontier-window2-vertexlist))
```

```
            (fw2-v2 (second frontier-window2-vertexlist))
            (fw2-v1-x (vertex-x-coord (eval fw2-v1)))
            (fw2-v1-y (vertex-y-coord (eval fw2-v1)))
            (fw2-v2-x (vertex-x-coord (eval fw2-v2)))
            (fw2-v2-y (vertex-y-coord (eval fw2-v2)))
            (cross-heading
              (if (intersect fw1-v1-x fw1-v1-y fw2-v2-x fw2-v2-y
                             fw1-v2-x fw1-v2-y fw2-v1-x fw2-v1-y) t nil))
            (permissible-headings
              (if cross-heading
                  (list (heading fw1-v1-x fw1-v1-y fw2-v2-x fw2-v2-y)
                        (heading fw1-v2-x fw1-v2-y fw2-v1-x fw2-v1-y))
                  (list (heading fw1-v1-x fw1-v1-y fw2-v1-x fw2-v1-y)
                        (heading fw1-v2-x fw1-v2-y fw2-v2-x fw2-v2-y))))
            (permissible-heading-1 (first permissible-headings))
            (permissible-heading-2 (second permissible-headings))
            (tolerance 0.01)
            (ordered-headings
              (if (equal-within-tolerance
                    permissible-heading-1 permissible-heading-2 tolerance)
                  nil
                  (order-headings permissible-heading-1 permissible-heading-2))))
        (if ordered-headings
            (list (list (first ordered-headings) 'OP)
                  (list (second ordered-headings) 'OP)))))


(defun permissible-heading-range-from-edge-to-edge-incident
       (frontier-window1 frontier-window2)
  (let* ((frontier-window1-vertexlist
           (edge-vertex-list (eval frontier-window1)))
         (fw1-v1 (first frontier-window1-vertexlist))
         (fw1-v2 (second frontier-window1-vertexlist))
         (frontier-window2-vertexlist
           (edge-vertex-list (eval frontier-window2)))
         (fw2-v1 (first frontier-window2-vertexlist))
         (fw2-v2 (second frontier-window2-vertexlist))
         (fw1-v1-fw2-v1 (if (vertex-equal-p fw1-v1 fw2-v1) t nil))
         (fw1-v1-fw2-v2 (if (vertex-equal-p fw1-v1 fw2-v2) t nil))
         (fw2-v1-fw1-v1 (if (vertex-equal-p fw2-v1 fw1-v1) t nil))
         (fw2-v1-fw1-v2 (if (vertex-equal-p fw2-v1 fw1-v2) t nil))
         (fw1-incident-vertex
           (if (or fw1-v1-fw2-v1 fw1-v1-fw2-v2) fw1-v1 fw1-v2))
         (fw1-incident-vertex-x (vertex-x-coord (eval fw1-incident-vertex)))
         (fw1-incident-vertex-y (vertex-y-coord (eval fw1-incident-vertex)))
         (fw1-non-incident-vertex
           (if (or fw1-v1-fw2-v1 fw1-v1-fw2-v2) fw1-v2 fw1-v1))
         (fw1-non-incident-vertex-x
           (vertex-x-coord (eval fw1-non-incident-vertex)))
         (fw1-non-incident-vertex-y
           (vertex-y-coord (eval fw1-non-incident-vertex)))
         (permissible-heading-1
           (heading fw1-non-incident-vertex-x fw1-non-incident-vertex-y
                    fw1-incident-vertex-x fw1-incident-vertex-y))
         (fw2-incident-vertex
           (if (or fw2-v1-fw1-v1 fw2-v1-fw1-v2) fw2-v1 fw2-v2))
         (fw2-incident-vertex-x (vertex-x-coord (eval fw2-incident-vertex)))
         (fw2-incident-vertex-y (vertex-y-coord (eval fw2-incident-vertex)))
         (fw2-non-incident-vertex
           (if (or fw2-v1-fw1-v1 fw2-v1-fw1-v2) fw2-v2 fw2-v1))
```

```
        (fw2-non-incident-vertex-x
          (vertex-x-coord (eval fw2-non-incident-vertex)))
        (fw2-non-incident-vertex-y
          (vertex-y-coord (eval fw2-non-incident-vertex)))
        (permissible-heading-2
          (heading fw2-incident-vertex-x fw2-incident-vertex-y
                   fw2-non-incident-vertex-x fw2-non-incident-vertex-y))
        (tolerance 0.01)
        (ordered-headings
          (if (equal-within-tolerance
                permissible-heading-1 permissible-heading-2 tolerance)
              nil
              (order-headings permissible-heading-1 permissible-heading-2))))
    (if ordered-headings
        (list (list (first ordered-headings) 'OP)
              (list (second ordered-headings) 'OP)))))


(defun permissible-heading-range-from-edge-to-edge-obscure
       (frontier-window1 frontier-window1-approach-region
        frontier-window2 frontier-window2-approach-region)
  (let* ((obscured-edge (obscure-edge frontier-window1 frontier-window2))
         (obscured-edge-approach-region
           (if (equal obscured-edge frontier-window1)
               frontier-window1-approach-region
               frontier-window2-approach-region))
         (obscuring-edge
           (if (equal obscured-edge frontier-window1)
               frontier-window2 frontier-window1))
         (obscuring-edge-approach-region
           (if (equal obscured-edge-approach-region
                frontier-window1-approach-region)
               frontier-window2-approach-region
               frontier-window1-approach-region))
         (obscured-edge-vertexlist (edge-vertex-list (eval obscured-edge)))
         (obscured-edge-v1 (first obscured-edge-vertexlist))
         (obscured-edge-v2 (second obscured-edge-vertexlist))
         (obscured-edge-v1-x (vertex-x-coord (eval obscured-edge-v1)))
         (obscured-edge-v1-y (vertex-y-coord (eval obscured-edge-v1)))
         (obscured-edge-v2-x (vertex-x-coord (eval obscured-edge-v2)))
         (obscured-edge-v2-y (vertex-y-coord (eval obscured-edge-v2)))
         (obscuring-edge-vertexlist (edge-vertex-list (eval obscuring-edge)))
         (obscuring-edge-v1 (first obscuring-edge-vertexlist))
         (obscuring-edge-v2 (second obscuring-edge-vertexlist))
         (obscuring-edge-v1-x (vertex-x-coord (eval obscuring-edge-v1)))
         (obscuring-edge-v1-y (vertex-y-coord (eval obscuring-edge-v1)))
         (obscuring-edge-v2-x (vertex-x-coord (eval obscuring-edge-v2)))
         (obscuring-edge-v2-y (vertex-y-coord (eval obscuring-edge-v2)))
         (obscuring-edge-line-equation
           (line-equation obscuring-edge-v1-x obscuring-edge-v1-y
                          obscuring-edge-v2-x obscuring-edge-v2-y))
         (distance-obscuring-edge-v1
           (distance-point-to-edge-min obscuring-edge-v1-x
                                       obscuring-edge-v1-y obscured-edge))
         (distance-obscuring-edge-v2
         (distance-point-to-edge-min obscuring-edge-v2-x
                                     obscuring-edge-v2-y obscured-edge))
         (closest-obscuring-edge-pt
           (if (<= distance-obscuring-edge-v1 distance-obscuring-edge-v2)
               obscuring-edge-v1 obscuring-edge-v2))
```

```
(closest-obscuring-edge-pt-x
  (if (equal closest-obscuring-edge-pt obscuring-edge-v1)
      obscuring-edge-v1-x obscuring-edge-v2-x))
(closest-obscuring-edge-pt-y
  (if (equal closest-obscuring-edge-pt obscuring-edge-v1)
      obscuring-edge-v1-y obscuring-edge-v2-y))
(farthest-obscuring-edge-pt
  (if (equal closest-obscuring-edge-pt obscuring-edge-v1)
      obscuring-edge-v2 obscuring-edge-v1))
(farthest-obscuring-edge-pt-x
  (if (equal farthest-obscuring-edge-pt obscuring-edge-v1)
      obscuring-edge-v1-x obscuring-edge-v2-x))
(farthest-obscuring-edge-pt-y
  (if (equal farthest-obscuring-edge-pt obscuring-edge-v1)
      obscuring-edge-v1-y obscuring-edge-v2-y))
(frontier-window1-obscured
  (if (equal obscured-edge frontier-window1) t nil))
(frontier-window2-obscured
  (if (equal obscured-edge frontier-window2) t nil))
(test-vertex
  (first (remove-items obscuring-edge-vertexlist
         (get-vertexlist-from-region obscuring-edge-approach-region))))
(test-vertex-x (vertex-x-coord (eval test-vertex)))
(test-vertex-y (vertex-y-coord (eval test-vertex)))
(le-solution-obscured-edge-v1
  (line-equation-solution obscuring-edge-line-equation
    obscured-edge-v1-x obscured-edge-v1-y))
(le-solution-test-vertex
  (line-equation-solution obscuring-edge-line-equation
    test-vertex-x test-vertex-y))
(obscured-edge-vertex
  (if frontier-window1-obscured
      (if (or (and (plusp le-solution-obscured-edge-v1)
                   (plusp le-solution-test-vertex))
              (and (minusp le-solution-obscured-edge-v1)
                   (minusp le-solution-test-vertex)))
          obscured-edge-v1 obscured-edge-v2)
      (if frontier-window2-obscured
          (if (or (and (plusp le-solution-obscured-edge-v1)
                       (minusp le-solution-test-vertex))
                  (and (minusp le-solution-obscured-edge-v1)
                       (plusp le-solution-test-vertex)))
              obscured-edge-v1 obscured-edge-v2))))
(obscured-edge-vertex-x
  (if (equal obscured-edge-vertex obscured-edge-v1)
      obscured-edge-v1-x obscured-edge-v2-x))
(obscured-edge-vertex-y
  (if (equal obscured-edge-vertex obscured-edge-v1)
      obscured-edge-v1-y obscured-edge-v2-y))
(permissible-heading-1
  (if frontier-window1-obscured
      (heading closest-obscuring-edge-pt-x closest-obscuring-edge-pt-y
               farthest-obscuring-edge-pt-x
               farthest-obscuring-edge-pt-y)
      (if frontier-window2-obscured
          (heading farthest-obscuring-edge-pt-x
                   farthest-obscuring-edge-pt-y
                   closest-obscuring-edge-pt-x
                   closest-obscuring-edge-pt-y))))
(permissible-heading-2
```

```
          (if frontier-window1-obscured
              (heading obscured-edge-vertex-x obscured-edge-vertex-y
                       closest-obscuring-edge-pt-x closest-obscuring-edge-pt-y)
              (if frontier-window2-obscured
                  (heading closest-obscuring-edge-pt-x
                           closest-obscuring-edge-pt-y
                           obscured-edge-vertex-x
                           obscured-edge-vertex-y))))
        (tolerance 0.01)
        (ordered-headings
          (if (equal-within-tolerance
                permissible-heading-1 permissible-heading-2 tolerance)
              nil
              (order-headings permissible-heading-1 permissible-heading-2))))
    (if ordered-headings
        (list (list (first ordered-headings) 'OP)
              (list (second ordered-headings) 'OP)))))


(defun permissible-heading-range-from-edge-to-edge
       (frontier-window1 frontier-window2)
  (if (member frontier-window1
              (get-incident-edgelist-from-edge frontier-window2))
      (permissible-heading-range-from-edge-to-edge-incident
        frontier-window1 frontier-window2)
      (permissible-heading-range-from-edge-to-edge-non-incident
        frontier-window1 frontier-window2)))


(defun permissible-heading-range-from-vertex-to-edge
       (frontier-window1 frontier-window2)
  (let* ((fw1-x (vertex-x-coord (eval frontier-window1)))
         (fw1-y (vertex-y-coord (eval frontier-window1)))
         (frontier-window2-vertexlist
           (edge-vertex-list (eval frontier-window2)))
         (fw2-v1 (first frontier-window2-vertexlist))
         (fw2-v2 (second frontier-window2-vertexlist))
         (fw2-v1-x (vertex-x-coord (eval fw2-v1)))
         (fw2-v1-y (vertex-y-coord (eval fw2-v1)))
         (fw2-v2-x (vertex-x-coord (eval fw2-v2)))
         (fw2-v2-y (vertex-y-coord (eval fw2-v2)))
         (permissible-heading-1 (heading fw1-x fw1-y fw2-v1-x fw2-v1-y))
         (permissible-heading-2 (heading fw1-x fw1-y fw2-v2-x fw2-v2-y))
         (tolerance 0.01)
         (ordered-headings
           (if (equal-within-tolerance
                 permissible-heading-1 permissible-heading-2 tolerance)
               nil
               (order-headings permissible-heading-1 permissible-heading-2))))
    (if ordered-headings
        (list (list (first ordered-headings) 'OP)
              (list (second ordered-headings) 'OP)))))


(defun permissible-heading-range-from-edge-to-vertex
       (frontier-window1 frontier-window2)
  (let* ((frontier-window1-vertexlist
           (edge-vertex-list (eval frontier-window1)))
         (fw1-v1 (first frontier-window1-vertexlist))
         (fw1-v2 (second frontier-window1-vertexlist))
```

```
          (fw1-v1-x (vertex-x-coord (eval fw1-v1)))
          (fw1-v1-y (vertex-y-coord (eval fw1-v1)))
          (fw1-v2-x (vertex-x-coord (eval fw1-v2)))
          (fw1-v2-y (vertex-y-coord (eval fw1-v2)))
          (fw2-x (vertex-x-coord (eval frontier-window2)))
          (fw2-y (vertex-y-coord (eval frontier-window2)))
          (permissible-heading-1 (heading fw1-v2-x fw1-v2-y fw2-x fw2-y))
          (permissible-heading-2 (heading fw1-v1-x fw1-v1-y fw2-x fw2-y))
          (tolerance 0.01)
          (ordered-headings
            (if (equal-within-tolerance
                  permissible-heading-1 permissible-heading-2 tolerance)
                nil
                (order-headings permissible-heading-1 permissible-heading-2))))
    (if ordered-headings
        (list (list (first ordered-headings) 'OP)
              (list (second ordered-headings) 'OP)))))


(defun permissible-heading-range-from-vertex-to-vertex
       (frontier-window1 frontier-window2)
  (let* ((fw2-x (vertex-x-coord (eval frontier-window2)))
         (fw2-y (vertex-y-coord (eval frontier-window2)))
         (fw1-x (vertex-x-coord (eval frontier-window1)))
         (fw1-y (vertex-y-coord (eval frontier-window1)))
         (permissible-heading (heading fw1-x fw1-y fw2-x fw2-y)))
    (list (list permissible-heading 'CL))))


;*****************************************************************************
;
; Path Planning Model: Construction of Permissible Headings
;
;*****************************************************************************


(defun permissible-headings-critical-stability (frontier-region)
  (let* ((stability-constraints
          (region-stability-constraints (eval frontier-region))))
    (if stability-constraints
        (let* ((ordered-headings-range1
                  (order-headings (first stability-constraints)
                                  (second stability-constraints)))
               (ordered-headings-range2
                 (order-headings (third stability-constraints)
                                 (fourth stability-constraints))))
          (list (list (list (first ordered-headings-range1) 'CL)
                      (list (second ordered-headings-range1) 'CL))
          (list (list (first ordered-headings-range2) 'CL)
                (list (second ordered-headings-range2) 'CL)))))))


(defun impermissible-headings-critical-instability (frontier-region)
  (let* ((stability-constraints
          (region-stability-constraints (eval frontier-region))))
    (if stability-constraints
        (let* ((ordered-headings-range1
                  (order-headings (first stability-constraints)
                                  (fourth stability-constraints)))
               (ordered-headings-range2
```

```
                (order-headings (second stability-constraints)
                                (third stability-constraints))))
          (list (list (list (first ordered-headings-range1) 'OP)
                      (list (second ordered-headings-range1) 'OP))
                (list (list (first ordered-headings-range2) 'OP)
                      (list (second ordered-headings-range2) 'OP)))))))


(defun permissible-headings-critical-braking (frontier-region)
  (let* ((braking-constraints
          (region-braking-constraints (eval frontier-region))))
    (if braking-constraints
        (let* ((ordered-headings-range
                (order-headings (first braking-constraints)
                                (second braking-constraints))))
          (list (list (list (first ordered-headings-range) 'CL)
                      (list (second ordered-headings-range) 'CL)))))))


(defun permissible-headings-geometric (frontier-window1 frontier-window2)
  (let ((frontier-window1-vertex (vertex-p (eval frontier-window1)))
        (frontier-window2-vertex (vertex-p (eval frontier-window2))))
    (if (and (null frontier-window1-vertex) (null frontier-window2-vertex))
        (list (permissible-heading-range-from-edge-to-edge
               frontier-window1 frontier-window2))
        (if (and frontier-window1-vertex (null frontier-window2-vertex))
            (list (permissible-heading-range-from-vertex-to-edge
                   frontier-window1 frontier-window2))
            (if (and (null frontier-window1-vertex) frontier-window2-vertex)
                (list (permissible-heading-range-from-edge-to-vertex
                       frontier-window1 frontier-window2))
                (if (and frontier-window1-vertex frontier-window2-vertex)
                    (list (permissible-heading-range-from-vertex-to-vertex
                           frontier-window1 frontier-window2)))))))))


(defun permissible-headings-stability (frontier-window1 frontier-window2)
  (let* ((frontier-window1-vertex (if (vertex-p (eval frontier-window1)) t nil))
         (frontier-window2-vertex (if (vertex-p (eval frontier-window2)) t nil))
         (post-frontier-region
           (if
             (and (null frontier-window1-vertex) (null frontier-window2-vertex))
             (get-traversal-region-edge-edge frontier-window1 frontier-window2)
             (if
               (and frontier-window1-vertex (null frontier-window2-vertex))
               (get-traversal-region-vertex-edge frontier-window1
                                                 frontier-window2)
               (if
                 (and (null frontier-window1-vertex) frontier-window2-vertex)
                 (get-traversal-region-edge-vertex frontier-window1
                                                   frontier-window2)
                 (if
                   (and frontier-window1-vertex frontier-window2-vertex)
                   (get-traversal-region-vertex-vertex frontier-window1
                                                       frontier-window2)))))))
    (if post-frontier-region
        (if (anisotropic-partially-safe-region-p post-frontier-region)
            (permissible-headings-intersection
              (permissible-headings-geometric frontier-window1 frontier-window2)
              (permissible-headings-critical-stability post-frontier-region))
```

```
                (permissible-headings-geometric frontier-window1
                                                frontier-window2)))))


(defun permissible-headings-braking (frontier-window1 frontier-window2)
  (let* ((frontier-window1-vertex (if (vertex-p (eval frontier-window1)) t nil))
         (frontier-window2-vertex (if (vertex-p (eval frontier-window2)) t nil))
         (post-frontier-region
           (if
             (and (null frontier-window1-vertex) (null frontier-window2-vertex))
             (get-traversal-region-edge-edge frontier-window1
                                             frontier-window2)
             (if
               (and frontier-window1-vertex (null frontier-window2-vertex))
               (get-traversal-region-vertex-edge frontier-window1
                                                 frontier-window2)
               (if
                 (and (null frontier-window1-vertex) frontier-window2-vertex)
                 (get-traversal-region-edge-vertex frontier-window1
                                                   frontier-window2)
                 (if
                   (and frontier-window1-vertex frontier-window2-vertex)
                   (get-traversal-region-vertex-vertex frontier-window1
                                                       frontier-window2)))))))
    (if post-frontier-region
        (permissible-headings-intersection
          (permissible-headings-stability frontier-window1 frontier-window2)
          (permissible-headings-critical-braking post-frontier-region)))))


(defun permissible-headings-non-braking (frontier-window1 frontier-window2)
  (let* ((frontier-window1-vertex (if (vertex-p (eval frontier-window1)) t nil))
         (frontier-window2-vertex (if (vertex-p (eval frontier-window2)) t nil))
         (post-frontier-region
           (if
             (and (null frontier-window1-vertex) (null frontier-window2-vertex))
             (get-traversal-region-edge-edge frontier-window1
                                             frontier-window2)
             (if
               (and frontier-window1-vertex (null frontier-window2-vertex))
               (get-traversal-region-vertex-edge frontier-window1
                                                 frontier-window2)
               (if
                 (and (null frontier-window1-vertex) frontier-window2-vertex)
                 (get-traversal-region-edge-vertex frontier-window1
                                                   frontier-window2)
                 (if
                   (and frontier-window1-vertex frontier-window2-vertex)
                   (get-traversal-region-vertex-vertex frontier-window1
                                                       frontier-window2)))))))
    (if post-frontier-region
        (let* ((ph-stability
                 (permissible-headings-stability frontier-window1
                                                 frontier-window2))
               (ph-critical-braking
                 (permissible-headings-critical-braking post-frontier-region))
               (stability-heading-list-1 (first (first ph-stability)))
               (stability-heading-1 (first stability-heading-list-1))
               (stability-heading-list-2 (second (first ph-stability)))
               (stability-heading-2 (first stability-heading-list-2))
```

```
     (braking-heading-list-1 (first (first ph-critical-braking)))
     (braking-heading-1 (first braking-heading-list-1))
     (braking-heading-list-2 (second (first ph-critical-braking)))
     (braking-heading-2 (first braking-heading-list-2))
     (tolerance 0.01))
(if
  (null (permissible-headings-braking frontier-window1
                                      frontier-window2))
  ph-stability
  (if
    (and (= (length (first ph-stability)) 1)
         (braking-heading-p stability-heading-1 post-frontier-region))
    nil
    (if
      (and (equal-within-tolerance braking-heading-1
                                   stability-heading-1 tolerance)
           (equal-within-tolerance braking-heading-2
                                   stability-heading-2 tolerance))
      nil
      (if
        (and (interval-heading-range-p braking-heading-list-1
                                       stability-heading-list-1
                                       stability-heading-list-2)
             (interval-heading-range-p braking-heading-list-2
                                       stability-heading-list-1
                                       stability-heading-list-2))
        (list (interval-order-headings
                stability-heading-list-1 (list braking-heading-1 'OP))
              (interval-order-headings
                (list braking-heading-2 'OP)
                stability-heading-list-2))
        (if
          (or (and (interval-heading-range-p braking-heading-list-1
                                             stability-heading-list-1
                                             stability-heading-list-2)
                   (null (interval-heading-range-p
                           braking-heading-list-2
                           stability-heading-list-1
                           stability-heading-list-2)))
              (and (interval-heading-range-p braking-heading-list-1
                                             stability-heading-list-1
                                             stability-heading-list-2)
                   (equal-within-tolerance
                     braking-heading-2 stability-heading-2
                     tolerance)))
          (list (interval-order-headings
                  stability-heading-list-1
                  (list braking-heading-1 'OP)))
          (if
            (or (and (interval-heading-range-p
                       braking-heading-list-2
                       stability-heading-list-1
                       stability-heading-list-2)
                     (null (interval-heading-range-p
                             braking-heading-list-1
                             stability-heading-list-1
                             stability-heading-list-2)))
                (and (interval-heading-range-p
                       braking-heading-list-2
                       stability-heading-list-1
```

```
                                    stability-heading-list-2)
                                  (equal-within-tolerance
                                    braking-heading-1 stability-heading-1
                                    tolerance)))
                            (list (interval-order-headings
                                    stability-heading-list-2
                                    (list braking-heading-2 'OP)))
                            (if
                              (or (and (null (interval-heading-range-p
                                               braking-heading-list-1
                                               stability-heading-list-1
                                               stability-heading-list-2))
                                       (equal-within-tolerance
                                         braking-heading-2 stability-heading-1
                                         tolerance))
                                  (and (null (interval-heading-range-p
                                               braking-heading-list-2
                                               stability-heading-list-1
                                               stability-heading-list-2))
                                       (equal-within-tolerance
                                         braking-heading-1 stability-heading-2
                                         tolerance))
                                  (and (null (interval-heading-range-p
                                               braking-heading-list-1
                                               stability-heading-list-1
                                               stability-heading-list-2))
                                       (null (interval-heading-range-p
                                               braking-heading-list-2
                                               stability-heading-list-1
                                               stability-heading-list-2))))
                            nil))))))))))))

(defun permissible-headings-intersection
       (permissible-headings-1 permissible-headings-2)
  (let ((permissible-headings nil))
    (dolist (permissible-heading-range1 permissible-headings-1)
      (dolist (permissible-heading-range2 permissible-headings-2)
        (setf permissible-headings
              (cons (interval-heading-range-intersection
                      permissible-heading-range1 permissible-heading-range2)
                    permissible-headings))))
    (remove-if 'null permissible-headings)))

;*****************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-SEARCH-UTILITIES-II
;
; Structures: SEARCH-NODE (window regions region-type traversal-type
;                          permissible-headings cost-from-start
;                          estimate-to-goal)
;
; Functions: BUILD-SEARCH-NODE (swindow sregion srtype sttype sperheadings cost
;                               estimate)
;            BUILD-INITIAL-AGENDA ()
;            EXPAND-FRONTIER (pre-frontier-search-node)
;            BUILD-FRONTIER-SEARCH-NODE-LIST
;              (pre-frontier-search-node post-frontier-expansion-list)
;            GET-BASE-WINDOW-AND-APPROACH-REGION (post-frontier-search-node)
;            EXPAND-FRONTIER-WINDOW
;              (frontier-window pre-frontier-traversal-type
;           pre-frontier-region permissible-headings)
;              EXPAND-FRONTIER-WINDOW-GENERIC
;               (frontier-window pre-frontier-region)
;              EXPAND-VERTEX-WINDOW-I
;               (frontier-window post-frontier-window post-frontier-region)
;              EXPAND-VERTEX-WINDOW-IV
;               (frontier-window post-frontier-window post-frontier-region)
;              EXPAND-EDGE-WINDOW-I-FROM-I
;               (frontier-window post-frontier-window
;               post-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-I-FROM-II
;               (frontier-window post-frontier-window post-frontier-region
;                pre-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-I-FROM-III
;               (frontier-window post-frontier-window
;               post-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-II-FROM-I
;               (frontier-window post-frontier-window
;                post-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-II-FROM-II
;               (frontier-window post-frontier-window
;                post-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-II-FROM-IV
;               (frontier-window post-frontier-window
;                post-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-IV-FROM-I
;               (frontier-window post-frontier-window
;                post-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-IV-FROM-II
;               (frontier-window post-frontier-window
;                post-frontier-region permissible-headings)
;              EXPAND-EDGE-WINDOW-IV-FROM-IV
;               (frontier-window post-frontier-window
;                post-frontier-region permissible-headings)
;
;
; Global Variables: *virtual-vertex-count*     {virtual vertex count}
;                   *virtual-edge-count*       {virtual edge count}
;                   *search-node-count*        {search node count}
;
;
;*****************************************************************************
```

```
;*****************************************************************************
;
; Path Planning Model: Definition of Primitive Structures
;
;*****************************************************************************


(defstruct search-node window region region-type traversal-type
          permissible-headings cost-from-start estimate-to-goal)


;*****************************************************************************
;
; Path Planning Model: Construction of Primitive Structures
;
;*****************************************************************************


(defun build-search-node (swindow sregion sregiontype straversaltype
              sperheadings scost sestimate)
  (make-search-node ':window swindow
                    ':region sregion
                    ':region-type sregiontype
                    ':traversal-type straversaltype
                    ':permissible-headings sperheadings
                    ':cost-from-start scost
                    ':estimate-to-goal sestimate))


;*****************************************************************************
;
; Path Planning Model: Construction of Initial Agenda
;
;*****************************************************************************


(defun build-initial-agenda ()
  (eval (list 'setf 'n-0
              (build-search-node
                '(s-v) nil nil nil nil '(0.0)
                (list
                 (* (distance-vertex-to-vertex 's-v 'g-v)
                    *motion-resistance-lower*)))))
  (setf *search-node-count* 1) '(n-0))


(defun expand-frontier (pre-frontier-search-node)
  (let ((post-frontier-search-node-list
          (build-frontier-search-node-list pre-frontier-search-node))
        (revised-agenda nil))
    (dolist (post-frontier-search-node post-frontier-search-node-list)
      (let* ((window-list
               (search-node-window (eval post-frontier-search-node)))
             (post-frontier-window (first window-list))
             (region-list
               (search-node-region (eval post-frontier-search-node)))
             (post-frontier-window-approach-region (first region-list))
             (traversal-type-list
               (search-node-traversal-type (eval post-frontier-search-node)))
             (post-frontier-traversal-type (first traversal-type-list))
```

```
      (pre-frontier-traversal-type (second traversal-type-list))
      (permissible-headings-list
        (search-node-permissible-headings
          (eval post-frontier-search-node)))
      (permissible-headings-post-frontier
        (first permissible-headings-list))
      (base-list
        (get-base-window-and-approach-region post-frontier-search-node))
      (base-window (first base-list))
      (base-window-approach-region (second base-list))
      (obscure-window
        (if (and (edge-p (eval base-window))
                 (edge-p (eval post-frontier-window)))
      (obscure-edge base-window post-frontier-window)))
      (base-window-obscure
        (if obscure-window
            (if (equal base-window obscure-window) t nil)))
      (post-frontier-window-obscure
        (if obscure-window
            (if (equal post-frontier-window obscure-window) t nil))))
(if (or (and (equal post-frontier-traversal-type 'I)
             (equal pre-frontier-traversal-type 'I))
        (and (equal post-frontier-traversal-type 'I)
             (equal pre-frontier-traversal-type 'I-B))
        (and (equal post-frontier-traversal-type 'IV)
             (equal pre-frontier-traversal-type 'IV))
        (and (equal post-frontier-traversal-type 'IV)
             (equal pre-frontier-traversal-type 'IV-B)))
    (let ((revised-permissible-headings nil))
      (if
        (and (vertex-p (eval base-window))
             (vertex-p (eval post-frontier-window)))
        (setf revised-permissible-headings
              (permissible-headings-intersection
                permissible-headings-post-frontier
                (list (permissible-heading-range-from-vertex-to-vertex
                        base-window post-frontier-window))))
        (if
          (and (vertex-p (eval base-window))
               (edge-p (eval post-frontier-window)))
          (setf revised-permissible-headings
                (permissible-headings-intersection
                  permissible-headings-post-frontier
                  (list (permissible-heading-range-from-vertex-to-edge
                          base-window post-frontier-window))))
          (if
            (and (edge-p (eval base-window))
                 (vertex-p (eval post-frontier-window)))
            (setf revised-permissible-headings
                  (permissible-headings-intersection
                    permissible-headings-post-frontier
                    (list (permissible-heading-range-from-edge-to-vertex
                            base-window post-frontier-window))))
            (if
              (and (edge-p (eval base-window))
                   (edge-p (eval post-frontier-window)))
              (if
                post-frontier-window-obscure
                (setf revised-permissible-headings
                  (permissible-headings-intersection
```

```
                                    permissible-headings-post-frontier
                                    (list
                                    (permissible-heading-range-from-edge-to-edge-obscure
                                      base-window base-window-approach-region
                                      post-frontier-window
                                      post-frontier-window-approach-region))))
                                (if
                                  base-window-obscure
                                  (setf revised-permissible-headings
                                    (permissible-headings-intersection
                                     permissible-headings-post-frontier
                                    (list
                                    (permissible-heading-range-from-edge-to-edge-obscure
                                      base-window base-window-approach-region
                                      post-frontier-window
                                      post-frontier-window-approach-region))))
                                  (if (and (null base-window-obscure)
                                           (null post-frontier-window-obscure))
                                    (setf revised-permissible-headings
                                      (permissible-headings-intersection
                                       permissible-headings-post-frontier
                                       (list
                                        (permissible-heading-range-from-edge-to-edge
                                          base-window post-frontier-window)))))))))))
                    (if revised-permissible-headings
                        (let ()
                          (setf (search-node-permissible-headings
                                  (eval post-frontier-search-node))
                                (cons revised-permissible-headings
                                      (rest (search-node-permissible-headings
                                              (eval post-frontier-search-node)))))
                          (setf revised-agenda
                                (cons post-frontier-search-node revised-agenda)))))
                  (setf revised-agenda
                        (cons post-frontier-search-node revised-agenda)))))
      revised-agenda))


(defun build-frontier-search-node-list (pre-frontier-search-node)
  (let* ((frontier-window
           (first (search-node-window (eval pre-frontier-search-node))))
         (pre-frontier-window
           (second (search-node-window (eval pre-frontier-search-node))))
         (pre-frontier-region
           (first (search-node-region (eval pre-frontier-search-node))))
         (pre-frontier-traversal-type
           (first (search-node-traversal-type (eval pre-frontier-search-node))))
         (permissible-headings
           (first (search-node-permissible-headings
                    (eval pre-frontier-search-node))))
         (post-frontier-expansion-list
           (expand-frontier-window frontier-window pre-frontier-traversal-type
                                   pre-frontier-region permissible-headings))
         (post-frontier-search-node-list nil))
    (dolist (candidate-window-list post-frontier-expansion-list)
      (let ((post-frontier-region (first candidate-window-list))
            (post-frontier-window (second candidate-window-list))
            (post-frontier-traversal-type (third candidate-window-list))
            (permissible-headings-post-frontier (fourth candidate-window-list))
            (permissible-headings-pre-frontier (fifth candidate-window-list)))
```

```
(if (member post-frontier-window
            (search-node-window (eval pre-frontier-search-node))) nil
    (let* ((designation (concat 'n- *search-node-count*))
           (window-list
             (cons post-frontier-window
               (search-node-window (eval pre-frontier-search-node))))
           (region-list
             (cons post-frontier-region
                   (search-node-region
                     (eval pre-frontier-search-node))))
           (region-type-list
             (cons
               (if
                 (isotropic-region-p post-frontier-region) 'is
                 (if
                   (anisotropic-safe-region-p post-frontier-region) 'as
                   (if
                     (anisotropic-partially-safe-region-p
                       post-frontier-region) 'ap)))
               (search-node-region-type
                 (eval pre-frontier-search-node))))
           (traversal-type-list
             (cons post-frontier-traversal-type
                   (search-node-traversal-type
                     (eval pre-frontier-search-node))))
           (permissible-headings-list
             (if permissible-headings-pre-frontier
                 (cons permissible-headings-post-frontier
                       (cons permissible-headings-pre-frontier
                             (rest (search-node-permissible-headings
                                     (eval pre-frontier-search-node)))))
                 (cons permissible-headings-post-frontier
                       (search-node-permissible-headings
                         (eval pre-frontier-search-node)))))
           (cost-rate *motion-resistance-lower*)
           (distance
             (if (and (edge-p (eval pre-frontier-window))
                      (edge-p (eval frontier-window))
                      (edge-p (eval post-frontier-window))
                      (tri-edge-incident-edge-p
                        pre-frontier-window frontier-window
                                            post-frontier-window))
           (distance-window-to-window-lower
             pre-frontier-window post-frontier-window)
           (distance-window-to-window-lower
             frontier-window post-frontier-window)))
           (cost-from-start-list
             (cons (+ (first (search-node-cost-from-start
                               (eval pre-frontier-search-node)))
                      (* cost-rate distance))
                   (search-node-cost-from-start
                     (eval pre-frontier-search-node))))
           (estimate-to-goal-list
             (cons (* cost-rate (distance-window-to-window-lower
                                  'g-v post-frontier-window))
                   (search-node-estimate-to-goal
                     (eval pre-frontier-search-node)))))
      (eval (list 'setf designation
                  (build-search-node window-list
                                     region-list
```

```
                                                region-type-list
                                                traversal-type-list
                                                permissible-headings-list
                                                cost-from-start-list
                                                estimate-to-goal-list)))
                    (setf *search-node-count* (1+ *search-node-count*))
                  (setf post-frontier-search-node-list
                        (cons designation post-frontier-search-node-list))))))
          post-frontier-search-node-list))


(defun get-base-window-and-approach-region (post-frontier-search-node)
  (let ((base-list nil))
    (do* ((window-list (rest (search-node-window
                                (eval post-frontier-search-node)))
                       (rest window-list))
          (region-list (search-node-region (eval post-frontier-search-node))
                       (rest region-list))
          (traversal-type-list (search-node-traversal-type
                                  (eval post-frontier-search-node))
                               (rest traversal-type-list)))
         ((not (null base-list)))
      (if (or (equal (first traversal-type-list) 'I-B)
              (equal (first traversal-type-list) 'IV-B))
          (setf base-list
                (cons (first window-list)
                      (cons (second region-list) base-list))))) base-list))


(defun expand-frontier-window (frontier-window pre-frontier-traversal-type
                               pre-frontier-region permissible-headings)
  (let* ((frontier-vertex-window (if (vertex-p (eval frontier-window)) t nil))
         (frontier-edge-window (if (edge-p (eval frontier-window)) t nil))
         (post-frontier-window-list
           (expand-frontier-window-generic frontier-window pre-frontier-region))
         (expansion-list nil))
    (dolist (post-frontier-window post-frontier-window-list)
      (let* ((post-frontier-vertex-window
        (if (vertex-p (eval post-frontier-window)) t nil))
            (post-frontier-edge-window
              (if (edge-p (eval post-frontier-window)) t nil))
            (post-frontier-region
              (if (and frontier-vertex-window post-frontier-vertex-window)
                  (get-traversal-region-vertex-vertex
                    frontier-window post-frontier-window)
                  (if (and frontier-vertex-window post-frontier-edge-window)
                      (get-traversal-region-vertex-edge
                        frontier-window post-frontier-window)
                      (if (and frontier-edge-window post-frontier-vertex-window)
                          (get-traversal-region-edge-vertex
                            frontier-window post-frontier-window)
                          (if (and frontier-edge-window
                                   post-frontier-edge-window)
                              (get-traversal-region-edge-edge
                                frontier-window pos  frontier-window)))))))
        (if post-frontier-region
            (let ((anisotropic-non-braking-region
                    (if (anisotropic-region-p post-frontier-region)
                        (if (permissible-headings-non-braking
                              frontier-window post-frontier-window) t nil)))
```

```
      (anisotropic-partially-safe-non-braking-region
        (if (anisotropic-partially-safe-region-p
              post-frontier-region)
            (if (permissible-headings-non-braking
                  frontier-window post-frontier-window) t nil)))
      (anisotropic-braking-region
        (if (anisotropic-region-p post-frontier-region)
            (if (permissible-headings-braking
                  frontier-window post-frontier-window) t nil))))
(if
  (or (isotropic-region-p post-frontier-region)
      anisotropic-non-braking-region)
      (if
        frontier-vertex-window
        (dolist (candidate-window-list
                  (expand-vertex-window-I
                    frontier-window post-frontier-window
                    post-frontier-region))
          (setf expansion-list
                (cons (cons post-frontier-region
                            candidate-window-list)
                      expansion-list)))
        (if
          (or (equal pre-frontier-traversal-type 'I)
              (equal pre-frontier-traversal-type 'I-B))
          (dolist (candidate-window-list
                    (expand-edge-window-I-from-I
                      frontier-window post-frontier-window
                      post-frontier-region permissible-headings))
            (setf expansion-list
                  (cons (cons post-frontier-region
                              candidate-window-list)
                        expansion-list)))
          (if
            (equal pre-frontier-traversal-type 'II)
            (dolist (candidate-window-list
                      (expand-edge-window-I-from-II
                        frontier-window
                        post-frontier-window post-frontier-region
                        pre-frontier-region permissible-headings))
              (setf expansion-list
                    (cons (cons post-frontier-region
                                candidate-window-list)
                          expansion-list)))
            (if
              (or (equal pre-frontier-traversal-type 'IV)
                  (equal pre-frontier-traversal-type 'IV-B))
              (dolist (candidate-window-list
                        (expand-edge-window-I-from-IV
                          frontier-window post-frontier-window
                          post-frontier-region
                          permissible-headings))
                (setf expansion-list
                      (cons (cons post-frontier-region
                                  candidate-window-list)
                            expansion-list)))))))))
(if
  anisotropic-braking-region
  (if
    frontier-vertex-window
```

```
    (dolist (candidate-window-list
              (expand-vertex-window-IV
                 frontier-window post-frontier-window
                 post-frontier-region))
      (setf expansion-list
            (cons (cons post-frontier-region
                        candidate-window-list)
                  expansion-list)))
    (if
      (or (equal pre-frontier-traversal-type 'I)
          (equal pre-frontier-traversal-type 'I-B))
      (dolist (candidate-window-list
                (expand-edge-window-IV-from-I
                  frontier-window post-frontier-window
                  post-frontier-region
                  permissible-headings))
        (setf expansion-list
              (cons (cons post-frontier-region
                          candidate-window-list)
                    expansion-list)))
      (if
        (equal pre-frontier-traversal-type 'II)
        (dolist (candidate-window-list
                  (expand-edge-window-IV-from-II
                    frontier-window
                    post-frontier-window
                    post-frontier-region
                    permissible-headings))
          (setf expansion-list
                (cons (cons post-frontier-region
                            candidate-window-list)
                      expansion-list)))
        (if
          (or (equal pre-frontier-traversal-type 'IV)
              (equal pre-frontier-traversal-type 'IV-B))
          (dolist (candidate-window-list
                    (expand-edge-window-IV-from-IV
                      frontier-window
                      post-frontier-window
                      post-frontier-region
                      permissible-headings))
            (setf expansion-list
                  (cons (cons post-frontier-region
                              candidate-window-list)
                        expansion-list)))))))))
(if
  anisotropic-partially-safe-non-braking-region
  (if
    (or (equal pre-frontier-traversal-type 'I)
        (equal pre-frontier-traversal-type 'I-B))
    (dolist (candidate-window-list
              (expand-edge-window-II-from-I
                frontier-window post-frontier-window
                post-frontier-region permissible-headings))
      (setf expansion-list
            (cons (cons post-frontier-region candidate-window-list)
                  expansion-list)))
    (if
      (equal pre-frontier-traversal-type 'II)
      (dolist (candidate-window-list
```

```
                              (expand-edge-window-II-from-I
                                frontier-window post-frontier-window
                                post-frontier-region permissible-headings))
                        (setf expansion-list
                              (cons (cons post-frontier-region
                                          candidate-window-list)
                                    expansion-list)))
                      (if
                        (or (equal pre-frontier-traversal-type 'IV)
                            (equal pre-frontier-traversal-type 'IV-B))
                        (dolist (candidate-window-list
                                  (expand-edge-window-II-from-I
                                    frontier-window post-frontier-window
                                    post-frontier-region permissible-headings))
                          (setf expansion-list
                                (cons (cons post-frontier-region
                                            candidate-window-list)
                                      expansion-list))))))))))))
    (remove-if '(lambda (candidate-window-list)
                  (if (null (second candidate-window-list)) t nil))
               expansion-list)))


(defun expand-frontier-window-generic (frontier-window pre-frontier-region)
  (if (vertex-p (eval frontier-window))
      (let* ((post-frontier-region-list
               (remove pre-frontier-region
                 (get-region-from-point
                   (vertex-x-coord (eval frontier-window))
                   (vertex-y-coord (eval frontier-window)))))
             (incident-edge-list (vertex-edge-list (eval frontier-window)))
             (post-frontier-region-edge-list
               (remove-duplicates
                 (remove-if
                   'boundary-edge-p
                     (remove-if
                       'obstacle-edge-p
                         (remove-items
                           incident-edge-list
                             (apply 'append
                               (mapcar 'region-edge-list
                                 (mapcar 'eval post-frontier-region-list))))))))
             (post-frontier-region-vertex-list
               (remove-duplicates
                 (remove-if 'boundary-vertex-p
                   (remove frontier-window
                     (apply 'append
                       (mapcar 'get-vertexlist-from-region
                         post-frontier-region-list)))))))
        (if (member 'g-v (vertex-visibility-list (eval frontier-window)))
            (list 'g-v)
            (append post-frontier-region-edge-list
                    post-frontier-region-vertex-list)))
      (if (edge-p (eval frontier-window))
          (let* ((post-frontier-region
                   (first (remove pre-frontier-region (edge-adjacency-list
                     (eval frontier-window)))))
                 (post-frontier-region-edge-list
                   (remove-if 'boundary-edge-p
                     (remove-if 'obstacle-edge-p
```

```
                    (remove frontier-window
                      (region-edge-list
                        (eval post-frontier-region))))))
                (post-frontier-region-vertex-list
                  (remove-if 'boundary-vertex-p
                    (remove-items (edge-vertex-list (eval frontier-window))
                      (get-vertexlist-from-region
                        post-frontier-region)))))
            (if (member 'g-v (edge-visibility-list (eval frontier-window)))
                (list 'g-v)
                (append post-frontier-region-edge-list
                        post-frontier-region-vertex-list))))))


(defun expand-vertex-window-I
       (frontier-window post-frontier-window post-frontier-region)
  (let ((permissible-headings-post-frontier
         (permissible-headings-non-braking frontier-window post-frontier-window))
        (candidate-window-lists nil))
    (if permissible-headings-post-frontier
        (dolist (permissible-heading-range-post-frontier
                  permissible-headings-post-frontier)
          (setf candidate-window-lists
                (cons (list post-frontier-window 'I-B
                            (list permissible-heading-range-post-frontier) nil)
                      candidate-window-lists))))  candidate-window-lists))


(defun expand-vertex-window-IV
       (frontier-window post-frontier-window post-frontier-region)
  (let ((permissible-headings-post-frontier
          (permissible-headings-braking frontier-window post-frontier-window))
        (candidate-window-lists nil))
    (if permissible-headings-post-frontier
        (setf candidate-window-lists
              (cons (list post-frontier-window 'IV-B
                          permissible-headings-post-frontier nil)
                    candidate-window-lists)))  candidate-window-lists))


(defun expand-edge-window-I-from-I
       (frontier-window post-frontier-window post-frontier-region
        permissible-headings)
  (let* ((permissible-headings-post-frontier
          (permissible-headings-intersection
            permissible-headings
            (permissible-headings-non-braking frontier-window
                                              post-frontier-windcw)))
         (candidate-window-lists nil))
    (if permissible-headings-post-frontier
        (dolist (permissible-heading-range-post-frontier
                  permissible-headings-post-frontier)
          (setf candidate-window-lists
                (cons (list post-frontier-window 'I
                            (list permissible-heading-range-post-frontier)
                            (list permissible-heading-range-post-frontier))
                      candidate-window-lists))))  candidate-window-lists))


(defun expand-edge-window-I-from-II
```

```
      (frontier-window post-frontier-window post-frontier-region
       pre-frontier-region permissible-headings)
  (let ((permissible-headings-post-frontier
          (permissible-headings-intersection
            (impermissible-headings-critical-instability pre-frontier-region)
            (permissible-headings-non-braking frontier-window
                                              post-frontier-window)))
        (candidate-window-lists nil))
    (if permissible-headings-post-frontier
        (dolist (permissible-heading-range-post-frontier
                 permissible-headings-post-frontier)
          (setf candidate-window-lists
                (cons (list post-frontier-window 'I-B
                            (list permissible-heading-range-post-frontier)
                            permissible-headings)
                      candidate-window-lists))))  candidate-window-lists))


(defun expand-edge-window-I-from-IV
       (frontier-window post-frontier-window post-frontier-region
       permissible-headings)
  (let* ((permissible-headings-pre-frontier permissible-headings)
        (frontier-window-vertexlist (edge-vertex-list (eval frontier-window)))
        (frontier-window-v1 (first frontier-window-vertexlist))
        (frontier-window-v1-x (vertex-x-coord (eval frontier-window-v1)))
        (frontier-window-v1-y (vertex-y-coord (eval frontier-window-v1)))
        (frontier-window-v2 (second frontier-window-vertexlist))
        (frontier-window-v2-x (vertex-x-coord (eval frontier-window-v2)))
        (frontier-window-v2-y (vertex-y-coord (eval frontier-window-v2)))
        (frontier-window-heading
          (heading frontier-window-v1-x frontier-window-v1-y
                   frontier-window-v2-x frontier-window-v2-y))
        (frontier-window-slope
          (if (region-orientation (eval post-frontier-region))
        (heading-inclination-angle
          frontier-window-heading
          (region-slope (eval post-frontier-region))
        (region-orientation (eval post-frontier-region))) 0.0))
        (adjusted-frontier-window-heading
          (if (plusp frontier-window-slope)
              (normalize-heading (+ frontier-window-heading 180.0))
              frontier-window-heading))
        (adjusted-frontier-window-slope
          (if (plusp frontier-window-slope) (- frontier-window-slope)
          frontier-window-slope))
        (frontier-window-heading-normal
          (if (permissible-headings-intersection
                (list (list (list
                  (+ adjusted-frontier-window-heading 90.0) 'CL)))
                  (permissible-headings-non-braking frontier-window
                                                    post-frontier-window))
              (normalize-heading (+ adjusted-frontier-window-heading 90.0))
              (normalize-heading (- adjusted-frontier-window-heading 90.0))))
        (optimal-braking-heading
          (abs (radians-to-degrees
            (asin (/ (- (tan (degrees-to-radians
              adjusted-frontier-window-slope)))
               *optimal-cost-rate*)))))
        (permissible-headings-optimal-braking
          (if (< optimal-braking-heading 90.0)
```

```
                (list
                  (list
                    (list (normalize-heading (+ frontier-window-heading-normal
                                                optimal-braking-heading)) 'CL)))))
            (permissible-headings-post-frontier
              (if permissible-headings-optimal-braking
                  (permissible-headings-intersection
                    permissible-headings-optimal-braking
                     (permissible-headings-non-braking frontier-window
                                                       post-frontier-window))))
            (candidate-window-lists nil))
          (if permissible-headings-post-frontier
              (setf candidate-window-lists
                    (cons (list post-frontier-window 'I-B
                                permissible-headings-post-frontier
                                permissible-headings-pre-frontier)
                          candidate-window-lists)))  candidate-window-lists))


(defun expand-edge-window-II-from-I
       (frontier-window post-frontier-window post-frontier-region
        permissible-headings)
  (let* ((stability-constraints
           (region-stability-constraints (eval post-frontier-region)))
         (permissible-headings-post-frontier
           (permissible-headings-intersection
             (mapcar '(lambda (stability-constraint)
                        (list (list stability-constraint 'CL)))
                     stability-constraints)
           (permissible-headings-non-braking
             frontier-window post-frontier-window)))
         (permissible-headings-pre-frontier
           (permissible-headings-intersection
             (impermissible-headings-critical-instability post-frontier-region)
              permissible-headings))
         (candidate-window-lists nil))
    (if permissible-headings-pre-frontier
        (dolist (permissible-heading-range-post-frontier
                  permissible-headings-post-frontier)
          (dolist (permissible-heading-range-pre-frontier
                    permissible-headings-pre-frontier)
            (setf candidate-window-lists
                  (cons (list post-frontier-window 'II
                              (list permissible-heading-range-post-frontier)
                              (list permissible-heading-range-pre-frontier))
                        candidate-window-lists)))))  candidate-window-lists))


(defun expand-edge-window-II-from-II
       (frontier-window post-frontier-window post-frontier-region
        pre-frontier-region permissible-headings)
  (let* ((stability-constraints
           (region-stability-constraints (eval post-frontier-region)))
           (permissible-headings-post-frontier
             (permissible-headings-intersection
               (mapcar '(lambda (stability-constraint)
                 (list (list stability-constraint 'CL)))
                         stability-constraints)
               (permissible-headings-non-braking
                 frontier-window post-frontier-window)))
```

```
        (candidate-window-lists nil))
    (if (permissible-headings-intersection
          (impermissible-headings-critical-instability post-frontier-region)
         permissible-headings)
        (dolist (permissible-heading-range-post-frontier
                 permissible-headings-post-frontier)
          (if (permissible-headings-intersection
                (impermissible-headings-critical-instability
                  pre-frontier-region)
                (list permissible-heading-range-post-frontier))
              (setf candidate-window-lists
                    (cons (list post-frontier-window 'II
                                (list permissible-heading-range-post-frontier)
                                permissible-headings)
                          candidate-window-lists)))))) candidate-window-lists))


(defun expand-frontier-window-II-from-IV
       (frontier-window post-frontier-window post-frontier-region
        permissible-headings)
  (let* ((stability-constraints (region-stability-constraints
           (eval post-frontier-region)))
         (permissible-headings-post-frontier
           (permissible-headings-intersection
             (mapcar '(lambda (stability-constraint)
               (list (list stability-constraint 'CL)))
                         stability-constraints)
             (permissible-headings-non-braking
               frontier-window post-frontier-window)))
         (frontier-window-vertexlist (edge-vertex-list (eval frontier-window)))
         (frontier-window-v1 (first frontier-window-vertexlist))
         (frontier-window-v1-x (vertex-x-coord (eval frontier-window-v1)))
         (frontier-window-v1-y (vertex-y-coord (eval frontier-window-v1)))
         (frontier-window-v2 (second frontier-window-vertexlist))
         (frontier-window-v2-x (vertex-x-coord (eval frontier-window-v2)))
         (frontier-window-v2-y (vertex-y-coord (eval frontier-window-v2)))
         (frontier-window-heading
           (heading frontier-window-v1-x frontier-window-v1-y
                    frontier-window-v2-x frontier-window-v2-y))
         (frontier-window-slope
           (if (region-orientation (eval post-frontier-region))
               (heading-inclination-angle
                 frontier-window-heading
                  (region-slope (eval post-frontier-region))
                    (region-orientation (eval post-frontier-region))) 0.0))
         (adjusted-frontier-window-heading
           (if (plusp frontier-window-slope)
               (normalize-heading (+ frontier-window-heading 180.0))
                 frontier-window-heading))
         (adjusted-frontier-window-slope
           (if (plusp frontier-window-slope) (- frontier-window-slope)
               frontier-window-slope))
         (frontier-window-heading-normal
           (if (permissible-headings-intersection
                 (list (list (list
                   (+ adjusted-frontier-window-heading 90.0) 'CL)))
                     (permissible-headings-non-braking frontier-window
                                                       post-frontier-window))
               (normalize-heading (+ adjusted-frontier-window-heading 90.0))
               (normalize-heading (- adjusted-frontier-window-heading 90.0))))))
```

```
            (optimal-braking-heading
              (abs (radians-to-degrees
                (asin (/ (- (tan (degrees-to-radians
                                  adjusted-frontier-window-slope)))
                    *optimal-cost-rate*)))))
            (permissible-headings-optimal-braking
              (if (< optimal-braking-heading 90.0)
                (list
                  (list
                    (list (normalize-heading (+ frontier-window-heading-normal
                      optimal-braking-heading)) 'CL)))))
            (candidate-window-lists nil))
        (if permissible-headings-optimal-braking
            (dolist (permissible-heading-range-post-frontier
                      permissible-headings-post-frontier)
              (if (permissible-headings-intersection
                    (impermissible-headings-critical-instability
                      post-frontier-region)
                      permissible-headings-optimal-braking)
                  (setf candidate-window-lists
                        (cons (list post-frontier-window 'II
                                    (list permissible-heading-range-post-frontier)
                                    permissible-headings)
                              candidate-window-lists))))) candidate-window-lists))


(defun expand-edge-window-IV-from-I
       (frontier-window post-frontier-window post-frontier-region
        permissible-headings)
  (let* ((permissible-headings-post-frontier
          (permissible-headings-braking frontier-window post-frontier-window))
          (frontier-window-vertexlist (edge-vertex-list (eval frontier-window)))
          (frontier-window-v1 (first frontier-window-vertexlist))
          (frontier-window-v1-x (vertex-x-coord (eval frontier-window-v1)))
          (frontier-window-v1-y (vertex-y-coord (eval frontier-window-v1)))
          (frontier-window-v2 (second frontier-window-vertexlist))
          (frontier-window-v2-x (vertex-x-coord (eval frontier-window-v2)))
          (frontier-window-v2-y (vertex-y-coord (eval frontier-window-v2)))
          (frontier-window-heading
            (heading frontier-window-v1-x frontier-window-v1-y
                     frontier-window-v2-x frontier-window-v2-y))
          (frontier-window-slope
            (if (region-orientation (eval post-frontier-region))
                (heading-inclination-angle
                  frontier-window-heading
                    (region-slope (eval post-frontier-region))
                      (region-orientation (eval post-frontier-region))) 0.0))
          (adjusted-frontier-window-heading
            (if (plusp frontier-window-slope)
                (normalize-heading (+ frontier-window-heading 180.0))
                  frontier-window-heading))
          (adjusted-frontier-window-slope
            (if (plusp frontier-window-slope) (- frontier-window-slope)
                frontier-window-slope))
          (frontier-window-heading-normal
            (if (permissible-headings-intersection
                  (list (list (list
                    (+ adjusted-frontier-window-heading 90.0) 'CL)))
                      (permissible-headings-braking frontier-window
                                                    post-frontier-window))
```

```
                (normalize-heading (+ adjusted-frontier-window-heading 90.0))
                (normalize-heading (- adjusted-frontier-window-heading 90.0))))
          (optimal-braking-heading
            (abs (radians-to-degrees
              (asin (/ (- (tan (degrees-to-radians
                adjusted-frontier-window-slope)))
                 *optimal-cost-rate*)))))
          (permissible-headings-optimal-braking
            (if (< optimal-braking-heading 90.0)
              (list
                (list
                  (list (normalize-heading (+ frontier-window-heading-normal
                                              optimal-braking-heading)) 'CL)))))
          (permissible-headings-pre-frontier
            (if permissible-headings-optimal-braking
                (permissible-headings-intersection
                  permissible-headings-optimal-braking
                  permissible-headings)))
          (candidate-window-lists nil))
      (if permissible-headings-pre-frontier
          (setf candidate-window-lists
                (cons (list post-frontier-window 'IV-B
                            permissible-headings-post-frontier
                            permissible-headings-pre-frontier)
                      candidate-window-lists)))  candidate-window-lists))


(defun expand-edge-window-IV-from-II
       (frontier-window post-frontier-window post-frontier-region
        permissible-headings)
  (let* ((permissible-headings-post-frontier
          (permissible-headings-braking frontier-window post-frontier-window))
          (frontier-window-vertexlist (edge-vertex-list (eval frontier-window)))
          (frontier-window-v1 (first frontier-window-vertexlist))
          (frontier-window-v1-x (vertex-x-coord (eval frontier-window-v1)))
          (frontier-window-v1-y (vertex-y-coord (eval frontier-window-v1)))
          (frontier-window-v2 (second frontier-window-vertexlist))
          (frontier-window-v2-x (vertex-x-coord (eval frontier-window-v2)))
          (frontier-window-v2-y (vertex-y-coord (eval frontier-window-v2)))
          (frontier-window-heading
            (heading frontier-window-v1-x frontier-window-v1-y
                     frontier-window-v2-x frontier-window-v2-y))
          (frontier-window-slope
            (if (region-orientation (eval post-frontier-region))
                (heading-inclination-angle
                  frontier-window-heading
                   (region-slope (eval post-frontier-region))
                     (region-orientation (eval post-frontier-region))) 0.0))
          (adjusted-frontier-window-heading
            (if (plusp frontier-window-slope)
                (normalize-heading (+ frontier-window-heading 180.0))
                 frontier-window-heading))
          (adjusted-frontier-window-slope
            (if (plusp frontier-window-slope) (- frontier-window-slope)
                frontier-window-slope))
          (frontier-window-heading-normal
            (if (permissible-headings-intersection
                  (list (list (list
                    (+ adjusted-frontier-window-heading 90.0) 'CL)))
                      (permissible-headings-braking frontier-window
```

```
                        post-frontier-window))
                  (normalize-heading (+ adjusted-frontier-window-heading 90.0))
                  (normalize-heading (- adjusted-frontier-window-heading 90.0))))
           (optimal-braking-heading
             (abs (radians-to-degrees
               (asin (/ (- (tan (degrees-to-radians
                 adjusted-frontier-window-slope)))
                  *optimal-cost-rate*)))))
           (permissible-headings-optimal-braking
             (if (< optimal-braking-heading 90.0)
                 (list
                   (list
                     (list (normalize-heading (+ frontier-window-heading-normal
                       optimal-braking-heading)) 'CL)))))
           (candidate-window-lists nil))
      (if permissible-headings-optimal-braking
          (if (permissible-headings-intersection
               (impermissible-headings-critical-instability post-frontier-region)
                permissible-headings-optimal-braking)
              (setf candidate-window-lists
                    (cons (list post-frontier-window 'IV-B
                                permissible-headings-post-frontier
                                permissible-headings)
                          candidate-window-lists))))  candidate-window-lists))


(defun expand-edge-window-IV-from-IV
       (frontier-window post-frontier-window post-frontier-region
        permissible-headings)
  (let* ((permissible-headings-post-frontier
          (permissible-headings-intersection
            permissible-headings
             (permissible-headings-braking
               frontier-window post-frontier-window)))
          (candidate-window-lists nil))
    (if permissible-headings-post-frontier
        (setf candidate-window-lists
              (cons (list post-frontier-window 'IV
                          permissible-headings-post-frontier
                          permissible-headings-post-frontier)
                    candidate-window-lists))
        (setf candidate-window-lists
              (cons (list post-frontier-window 'IV-B
                          (permissible-headings-braking
                            frontier-window post-frontier-window)
                           permissible-headings)
                    candidate-window-lists))) candidate-window-lists))


;*****************************************************************************
;
; Path-Planning Model: Definition and Initialization of Global Variables
;
;*****************************************************************************


(defvar *virtual-vertex-count*)
(defvar *virtual-edge-count*)
(defvar *search-node-count*)
```

```
(setf *virtual-vertex-count* 0)
(setf *virtual-edge-count* 0)
(setf *search-node-count* 0)


;*******************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-SEARCH-UTILITIES-III
;
; Functions: MPP ()
;            PLAN-PATH ()
;            DECOMPOSE-FEASIBLE-WINDOW-LIST (search-node)
;            EXTRACT-FEASIBLE-WINDOW-SUBLIST (search-node)
;            CONSOLIDATE-ISOTROPIC-AND-ANISOTROPIC-CORRIDORS (search-node)
;            DECOMPOSE-FEASIBLE-WINDOW-SUBLIST (search-node)
;            EXTRACT-DETERMINISTIC-ENTRY-PATH-SEGMENTS (search-node)
;            EXTRACT-DETERMINISTIC-EXIT-PATH-SEGMENTS (search-node)
;            GENERATE-OPTIMAL-PATH-SEGMENTS (search-node-list)
;            GENERATE-OPTIMAL-PATH-SEGMENTS-WITHIN-CORRIDOR (search-node)
;            GENERATE-OPTIMAL-PATH-SEGMENT-WITHIN-CORRIDOR
;              (start-point edge traversal-type-list permissible-headings-list)
;            GENERATE-PATH-SEGMENTS-WITHIN-CORRIDOR
;              (start-point way-point-list window-list traversal-type-list
;          permissible-headings-list)
;            SYNTHESIZE-OPTIMAL-PATH-SEGMENTS (optimal-path-segments)
;            BUILD-SEARCH-CORRIDOR (search-node)
;            GENERATE-OPTIMIZATION-WINDOWS
;              (start-point goal-point window-coord-lists
;               permissible-headings-list)
;            GENERATE-WAY-POINT
;              (start-point optimization-edge-segment permissible-headings)
;            GENERATE-WAY-POINT-DETERMINISTIC
;              (start-point edge-point1 edge-point2 permissible-headings)
;            GENERATE-WAY-POINT-NON-DETERMINISTIC (edge-point1 edge-point2)
;            PATH-SEGMENT-DETERMINISTIC-P (search-node)
;            PATH-SEGMENT-NON-DETERMINISTIC-P (search-node)
;            PURSUIT-EDGE-SEGMENT-POINT-EQUAL-P
;              (pursuit-edge-segment-pt1 pursuit-edge-segment-pt2)
;            POINT-IN-RANGE-P (target-point point1 point2 heading1 heading2)
;            POINT-IN-RANGE-SIMPLE-P (target-point point heading1 heading2)
;            POINT-IN-CORRIDOR-P (target-point point1 point2 heading)
;            LEAST-COST-P(search-node1 search-node2)
;            PATH-LENGTH (path)
;
; Global Variables: *agenda-length*
;                   *local-optimal-path-cost*
;                   *local-optimal-path*
;                   *total-local-optimal-paths*
;                   *global-optimal-path-cost*
;                   *global-optimal-path*
;
;*****************************************************************************


;*****************************************************************************
;
; Path Planning Model: Minimum-energy Path Planning (MPP) Algorithm
;
;*****************************************************************************


(defun plan-path ()
  (if (mission-go)
    (let ((universal-start-time (get-universal-time)))
      (retract-start-goal-visibility)
```

```
      (assert-start-goal-visibility)
      (do* ((revised-agenda nil
              (append candidate-search-node-list (rest agenda)))
            (agenda (build-initial-agenda) (sort revised-agenda 'least-cost-p))
            (candidate-search-node-list nil nil))
           ((null agenda) (setf *agenda-length* (length agenda)))
        (dolist (search-node (expand-frontier (first agenda)))
          (if (equal (first (search-node-window (eval search-node))) 'g-v)
              (let* ((search-node-list
                       (decompose-feasible-window-list search-node))
                     (local-optimal-path-segments
                       (if search-node-list
                           (generate-optimal-path-segments search-node-list)))
                     (local-optimal-path-list
                       (if local-optimal-path-segments
                           (synthesize-optimal-path-segments
                            local-optimal-path-segments))))
                (if local-optimal-path-segments
                    (let ()
                      (setf *agenda-length* (length agenda))
                      (setf *local-optimal-path*
                            (mapcar '(lambda (coord-list)
                                       (remove (third coord-list) coord-list))
                                    (second local-optimal-path-list)))
                      (setf *local-optimal-path-cost*
                            (first local-optimal-path-list))
                      (setf *total-local-optimal-paths*
                            (1+ *total-local-optimal-paths*))
                      (if (or (null *global-optimal-path*)
                              (< *local-optimal-path-cost*
                                 *global-optimal-path-cost*))
                          (let ()
                            (setf *global-optimal-path* *local-optimal-path*)
                            (setf *global-optimal-path-cost*
                                  *local-optimal-path-cost*)))
                      (refresh-display-with-local-path))))
              (let ((search-node-cost-estimate-current
                      (+ (first (search-node-cost-from-start
                                 (eval search-node)))
                         (first (search-node-estimate-to-goal
                                 (eval search-node))))))
                (if *global-optimal-path-cost*
                    (if (< search-node-cost-estimate-current
                           *global-optimal-path-cost*)
                        (setf candidate-search-node-list
                              (cons search-node candidate-search-node-list)))
                    (setf candidate-search-node-list
                          (cons search-node candidate-search-node-list)))))))
      (setf *global-planning-time*
            (- (get-universal-time) universal-start-time))
      (refresh-display-with-global-path)
      (special-effects))))


(defun MPP ()
  (retract-start-goal-visibility)
  (assert-start-goal-visibility)
  (terpri)
  (do* ((revised-agenda nil (append candidate-search-node-list (rest agenda)))
        (agenda (build-initial-agenda) (sort revised-agenda 'least-cost-p))
```

```
    (candidate-search-node-list nil nil)
    (global-optimal-path nil global-optimal-path)
    (global-optimal-path-cost nil global-optimal-path-cost))
   ((null agenda) (list global-optimal-path-cost global-optimal-path))
(terpri)
(princ "GLOBAL OPTIMAL PATH: ")
(princ global-optimal-path) (terpri)
(princ "GLOBAL OPTIMAL PATH COST: ")
(princ global-optimal-path-cost) (terpri)
(terpri)
(princ "CURRENT AGENDA: ")
(princ agenda) (terpri)
(dolist (search-node (expand-frontier (first agenda)))
  (if
    (equal (first (search-node-window (eval search-node))) 'g-v)
    (let* ((search-node-list
              (decompose-feasible-window-list search-node))
           (local-optimal-path-segments
             (if search-node-list
                 (generate-optimal-path-segments search-node-list)))
           (local-optimal-path-list
             (if local-optimal-path-segments
                 (synthesize-optimal-path-segments
                   local-optimal-path-segments)))
           (local-optimal-path
             (if local-optimal-path-segments
                 (mapcar '(lambda (coord-list)
                           (remove (third coord-list) coord-list))
                         (second local-optimal-path-list))))
           (local-optimal-path-cost
             (if local-optimal-path-segments
                 (first local-optimal-path-list))))
      (if local-optimal-path-list
           (if (or (null global-optimal-path)
                   (< local-optimal-path-cost global-optimal-path-cost))
               (let ()
                 (setf global-optimal-path local-optimal-path)
                 (setf global-optimal-path-cost local-optimal-path-cost))))
      (terpri)
      (princ "Pursuing Local Optimal Path...") (terpri) (terpri)
      (princ "FEASIBLE WINDOW LIST: ")
      (princ search-node) (terpri)
      (princ "FEASIBLE WINDOW SUBLIST: ")
      (princ search-node-list) (terpri) (terpri)
      (terpri)
      (princ "LOCAL OPTIMAL PATH SEGMENTS: ")
      (princ local-optimal-path-segments) (terpri)
      (princ "LOCAL OPTIMAL PATH: ")
      (princ local-optimal-path) (terpri)
      (princ "LOCAL OPTIMAL PATH COST: ")
      (princ local-optimal-path-cost) (terpri))
    (let ((search-node-cost-estimate-current
            (+ (first (search-node-cost-from-start (eval search-node)))
               (first (search-node-estimate-to-goal (eval search-node))))))
      (if global-optimal-path-cost
          (if (< search-node-cost-estimate-current global-optimal-path-cost)
              (setf candidate-search-node-list
                    (cons search-node candidate-search-node-list)))
          (setf candidate-search-node-list
                (cons search-node candidate-search-node-list)))))))
```

```
(defun decompose-feasible-window-list (search-node)
  (do* ((search-node-list
         (mapcar 'consolidate-isotropic-and-anisotropic-corridors
                 (extract-feasible-window-sublist search-node))
         (rest search-node-list))
        (pursuit-flag t pursuit-flag)
        (feasible-window-sublist
         (if (path-segment-deterministic-p (first search-node-list))
             (list (first search-node-list))
             (let ((feasible-window-sublist-result
                    (decompose-feasible-window-sublist
                      (first search-node-list))))
               (if feasible-window-sublist-result
                   feasible-window-sublist-result
                   (setf pursuit-flag nil))))
         (if (path-segment-deterministic-p (first search-node-list))
             (cons (first search-node-list) feasible-window-sublist)
             (let ((feasible-window-sublist-result
                    (decompose-feasible-window-sublist
                      (first search-node-list))))
               (if feasible-window-sublist-result
                   (append feasible-window-sublist-result
                           feasible-window-sublist)
                   (setf pursuit-flag nil))))))
       ((or (= (length search-node-list) 1) (null pursuit-flag))
        (if (null pursuit-flag) nil feasible-window-sublist))))


(defun extract-feasible-window-sublist (search-node)
  (let ((designation nil)
        (initial-vertex-flag t)
        (search-node-count 1)
        (feasible-window-sublist nil))
    (do ((window-list (reverse (search-node-window (eval search-node)))
                      (rest window-list))
         (region-list (reverse (search-node-region (eval search-node)))
                      (rest region-list))
         (region-type-list
           (reverse (search-node-region-type (eval search-node)))
           (rest region-type-list))
         (traversal-type-list
           (reverse (search-node-traversal-type (eval search-node)))
           (rest traversal-type-list))
         (permissible-headings-list
           (reverse (search-node-permissible-headings (eval search-node)))
           (rest permissible-headings-list)))
        ((null window-list))
      (if initial-vertex-flag
          (let ()
            (setf designation (concat (concat search-node '-)
                  search-node-count))
            (eval (list 'setf designation
                        (build-search-node
                          (list (first window-list))
                          (list (first region-list))
                          (list (first region-type-list))
                          (list (first traversal-type-list))
                          (list (first permissible-headings-list)) nil nil)))
            (setf initial-vertex-flag nil)
            (setf search-node-count (1+ search-node-count)))
```

```
            (if (and (null initial-vertex-flag)
                     (edge-p (eval (first window-list))))
                (let ()
                  (setf (search-node-window (eval designation))
                        (cons (first window-list)
                              (search-node-window (eval designation))))
                  (setf (search-node-region (eval designation))
                        (cons (first region-list)
                              (search-node-region (eval designation))))
                  (setf (search-node-region-type (eval designation))
                        (cons (first region-type-list)
                              (search-node-region-type (eval designation))))
                  (setf (search-node-traversal-type (eval designation))
                        (cons (first traversal-type-list)
                              (search-node-traversal-type (eval designation))))
                  (setf (search-node-permissible-headings (eval designation))
                        (cons (first permissible-headings-list)
                              (search-node-permissible-headings
                                (eval designation)))))
                (if (and (null initial-vertex-flag)
                         (vertex-p (eval (first window-list))))
                    (let ()
                      (setf (search-node-window (eval designation))
                            (cons (first window-list)
                                  (search-node-window (eval designation))))
                      (setf feasible-window-sublist
                            (cons designation feasible-window-sublist))
                      (if (null (equal (first window-list) 'g-v))
                          (let ()
                            (setf designation
                                  (concat (concat search-node '-)
                                          search-node-count))
                            (eval (list 'setf designation
                                        (build-search-node
                                          (list (first window-list))
                                          (list (first region-list))
                                          (list (first region-type-list))
                                          (list (first traversal-type-list))
                                          (list (first permissible-headings-list))
                                          nil nil)))
                            (setf search-node-count (1+ search-node-count)))))))))
    feasible-window-sublist))


(defun consolidate-isotropic-and-anisotropic-corridors (search-node)
  (let ((revised-window-list nil)
        (revised-traversal-type-list nil)
        (revised-permissible-headings-list nil)
        (first-window-in-corridor t))
    (do* ((window-list (search-node-window (eval search-node))
                       (rest window-list))
          (traversal-type-list (search-node-traversal-type (eval search-node))
                               (rest traversal-type-list))
          (traversal-type-I-base
            (if (equal (first traversal-type-list) 'I-B) t nil)
            (if (equal (first traversal-type-list) 'I-B) t nil))
          (traversal-type-I (if (equal (first traversal-type-list) 'I) t nil)
            (if (equal (first traversal-type-list) 'I) t nil))
          (traversal-type-IV-base
            (if (equal (first traversal-type-list) 'IV-B) t nil)
```

```
            (if (equal (first traversal-type-list) 'IV-B) t nil))
          (traversal-type-IV (if (equal (first traversal-type-list) 'IV) t nil)
            (if (equal (first traversal-type-list) 'IV) t nil))
          (permissible-headings-list (search-node-permissible-headings
                                       (eval search-node))
                                     (rest permissible-headings-list)))
       ((null traversal-type-list))
     (if first-window-in-corridor
         (let ((traversal-type
                 (if traversal-type-I-base 'I
                     (if traversal-type-IV-base 'IV
                         (first traversal-type-list)))))
           (if (or traversal-type-I traversal-type-IV)
               (setf first-window-in-corridor nil))
           (setf revised-window-list
                 (cons (first window-list) revised-window-list))
           (setf revised-traversal-type-list
                 (cons traversal-type revised-traversal-type-list))
           (setf revised-permissible-headings-list
                 (cons (first permissible-headings-list)
                       revised-permissible-headings-list))
           (if (vertex-p (eval (second window-list)))
               (setf revised-window-list
                     (cons (second window-list) revised-window-list))))
         (if (null first-window-in-corridor)
             (let ()
               (if (or traversal-type-I-base traversal-type-IV-base)
                   (setf first-window-in-corridor t))
               (if (vertex-p (eval (second window-list)))
                   (setf revised-window-list
                         (cons (second window-list) revised-window-list)))))))
   (setf (search-node-window (eval search-node)) revised-window-list)
   (setf (search-node-region (eval search-node)) nil)
   (setf (search-node-region-type (eval search-node)) nil)
   (setf (search-node-traversal-type (eval search-node))
                                     revised-traversal-type-list)
   (setf (search-node-permissible-headings (eval search-node))
                                           revised-permissible-headings-list)
   (setf (search-node-cost-from-start (eval search-node)) nil)
   (setf (search-node-estimate-to-goal (eval search-node)) nil) search-node))


(defun decompose-feasible-window-sublist (search-node)
  (let ((feasible-window-sublist nil)
        (feasible-window-sublist-entry
          (extract-deterministic-entry-path-segments search-node)))
    (if feasible-window-sublist-entry
        (let ((terminal-search-node
                (first (last feasible-window-sublist-entry))))
          (if (path-segment-deterministic-p terminal-search-node)
              (setf feasible-window-sublist feasible-window-sublist-entry)
              (let ((entry-search-nodes
                      (remove terminal-search-node
                        feasible-window-sublist-entry))
                    (feasible-window-sublist-exit
                      (extract-deterministic-exit-path-segments
                        terminal-search-node)))
                (if feasible-window-sublist-exit
                    (let ((exit-search-nodes
                            (rest feasible-window-sublist-exit))
```

```
                                (non-deterministic-search-node
                                  (first feasible-window-sublist-exit)))
                              (setf feasible-window-sublist
                                    (append entry-search-nodes
                                            (list non-deterministic-search-node)
                                            exit-search-nodes))))))))) feasible-window-sublist))


(defun extract-deterministic-entry-path-segments (search-node)
  (do* ((window-list
         (search-node-window (eval search-node))
           (cons virtual-vertex (rest (rest window-list))))
        (start-point (list (vertex-x-coord (eval (first window-list)))
                           (vertex-y-coord (eval (first window-list)))
                           (vertex-z-coord (eval (first window-list))))
                     way-point)
        (edge-points
          (if (edge-p (eval (second window-list)))
              (mapcar 'get-xyz-coord-from-vertex
                      (edge-vertex-list (eval (second window-list))))
              (get-xyz-coord-from-vertex (second window-list)))
          (if (edge-p (eval (second window-list)))
              (mapcar 'get-xyz-coord-from-vertex
                      (edge-vertex-list (eval (second window-list))))
              (get-xyz-coord-from-vertex (second window-list))))
        (traversal-type-list
          (search-node-traversal-type (eval search-node))
                                      (rest traversal-type-list))
        (permissible-headings-list
          (search-node-permissible-headings (eval search-node))
                                            (rest permissible-headings-list))
        (permissible-headings
          (if (edge-p (eval (second window-list)))
              (first permissible-headings-list)
              (list
                (list
                  (list
                    (heading (first start-point) (second start-point)
                             (first edge-points) (second edge-points))'cl))))
          (if (edge-p (eval (second window-list)))
              (first permissible-headings-list)
              (list
                (list
                  (list
                    (heading (first start-point) (second start-point)
                             (first edge-points) (second edge-points)) 'cl)))))
        (deterministic-approach-heading
          (if (= (length (first permissible-headings)) 1) t nil)
          (if (= (length (first permissible-headings)) 1) t nil))
        (pursuit-flag t pursuit-flag)
        (way-point
          (if (and deterministic-approach-heading
                   (edge-p (eval (second window-list))))
              (let ((way-point-result
                      (generate-way-point-deterministic
                        start-point (first edge-points) (second edge-points)
                         permissible-headings)))
                (if way-point-result way-point-result (setf pursuit-flag nil))))
          (if (and deterministic-approach-heading
                   (edge-p (eval (second window-list))))
```

```
            (let ((way-point-result
                   (generate-way-point-deterministic
                     start-point (first edge-points) (second edge-points)
                      permissible-headings)))
              (if way-point-result way-point-result
                                  (setf pursuit-flag nil)))))
        (virtual-vertex
          (if way-point
              (build-virtual-vertex
                (first way-point) (second way-point) (third way-point)
                  (second window-list)))
          (if way-point
              (build-virtual-vertex
                (first way-point) (second way-point) (third way-point)
                  (second window-list))))
        (search-node-count 1 (1+ search-node-count))
        (designation (concat (concat search-node '-) search-node-count)
                     (concat (concat search-node '-) search-node-count))
        (feasible-window-sublist nil feasible-window-sublist))
       ((or (null deterministic-approach-heading)
            (null pursuit-flag)
            (vertex-p (eval (second window-list))))
        (if (null pursuit-flag) nil
            (let ()
              (eval (list 'setf designation
                          (build-search-node
                            window-list nil nil
                            traversal-type-list
                            (if (edge-p (eval (second window-list)))
                                permissible-headings-list
                                (list permissible-headings))
                            nil nil)))
              (reverse (cons designation feasible-window-sublist)))))
    (eval (list 'setf designation
                (build-search-node
                  (list (first window-list) virtual-vertex) nil nil
                  (list (first traversal-type-list))
                  (list (first permissible-headings-list)) nil nil)))
    (setf feasible-window-sublist (cons designation feasible-window-sublist))))


(defun extract-deterministic-exit-path-segments (search-node)
  (do* ((window-list
          (reverse (search-node-window (eval search-node)))
          (cons virtual-vertex (rest (rest window-list))))
        (start-point (list (vertex-x-coord (eval (first window-list)))
                           (vertex-y-coord (eval (first window-list)))
                           (vertex-z-coord (eval (first window-list))))
                           way-point)
        (edge-points
          (if (edge-p (eval (second window-list)))
              (mapcar 'get-xyz-coord-from-vertex
                      (edge-vertex-list (eval (second window-list))))
              (get-xyz-coord-from-vertex (second window-list)))
          (if (edge-p (eval (second window-list)))
              (mapcar 'get-xyz-coord-from-vertex
                      (edge-vertex-list (eval (second window-list))))
              (get-xyz-coord-from-vertex (second window-list))))
        (traversal-type-list
          (reverse (search-node-traversal-type (eval search-node)))
```

```
          (rest traversal-type-list))
(permissible-headings-list
  (reverse (search-node-permissible-headings (eval search-node)))
          (rest permissible-headings-list))
(permissible-headings
  (if (edge-p (eval (second window-list)))
      (first permissible-headings-list)
      (list
        (list
          (list
            (heading (first edge-points) (second edge-points)
                     (first start-point) (second start-point)) 'cl))))
  (if (edge-p (eval (second window-list)))
      (first permissible-headings-list)
      (list
        (list
          (list
            (heading (first edge-points) (second edge-points)
                     (first start-point) (second start-point)) 'cl)))))
(deterministic-approach-heading
  (if (= (length (first permissible-headings)) 1) t nil)
  (if (= (length (first permissible-headings)) 1) t nil))
(pursuit-flag t pursuit-flag)
(revised-permissible-headings
  (if deterministic-approach-heading
      (list (list (list (reverse-heading
                          (first (first (first
                            permissible-headings)))) 'cl))))
  (if deterministic-approach-heading
      (list (list (list (reverse-heading
                          (first (first (first
                            permissible-headings)))) 'cl)))))
(way-point
  (if (and deterministic-approach-heading
           (edge-p (eval (second window-list))))
      (let ((way-point-result
              (generate-way-point-deterministic
                start-point (first edge-points) (second edge-points)
                 revised-permissible-headings)))
        (if way-point-result way-point-result
            (setf pursuit-flag nil))))
  (if (and deterministic-approach-heading
           (edge-p (eval (second window-list))))
      (let ((way-point-result
              (generate-way-point-deterministic
                start-point (first edge-points) (second edge-points)
                 revised-permissible-headings)))
        (if way-point-result way-point-result
            (setf pursuit-flag nil)))))
(virtual-vertex
  (if way-point
      (build-virtual-vertex (first way-point) (second way-point)
        (third way-point)
          (second window-list)))
  (if way-point
      (build-virtual-vertex (first way-point) (second way-point)
        (third way-point)
          (second window-list))))
(search-node-count 1 (1+ search-node-count))
(designation (concat (concat search-node '-) search-node-count)
```

```
                    (concat (concat search-node '-) search-node-count))
        (feasible-window-sublist nil feasible-window-sublist))
       ((or (null deterministic-approach-heading)
            (null pursuit-flag)
            (vertex-p (eval (second window-list))))
        (if (null pursuit-flag) nil
            (let ()
              (eval (list 'setf designation
                          (build-search-node
                            (reverse window-list) nil nil
                            (reverse traversal-type-list)
                            (if (edge-p (eval (second window-list)))
                                (reverse permissible-headings-list)
                                (list permissible-headings))
                           nil nil)))
              (cons designation feasible-window-sublist))))
    (eval (list 'setf designation
                (build-search-node
                  (list virtual-vertex (first window-list)) nil nil
                  (list (first traversal-type-list))
                  (list (first permissible-headings-list)) nil nil)))
    (setf feasible-window-sublist (cons designation feasible-window-sublist))))


;*******************************************************************************


(defun generate-optimal-path-segments (search-node-list)
  (do* ((optimal-path-segments nil optimal-path-segments)
        (optimization-search-node-list
          search-node-list (rest optimization-search-node-list))
        (search-node (first optimization-search-node-list)
          (first optimization-search-node-list))
        (traversal-type
          (if (path-segment-deterministic-p search-node)
              (first (search-node-traversal-type (eval search-node))))
          (if (path-segment-deterministic-p search-node)
              (first (search-node-traversal-type (eval search-node)))))
        (coord-list
          (if (path-segment-deterministic-p search-node)
              (mapcar 'get-xyz-coord-from-vertex
                (search-node-window (eval search-node))))
          (if (path-segment-deterministic-p search-node)
              (mapcar 'get-xyz-coord-from-vertex
                (search-node-window (eval search-node)))))
        (pursuit-flag t pursuit-flag)
        (optimal-path-segment
          (if (path-segment-deterministic-p search-node)
              (list traversal-type coord-list))
          (if (path-segment-deterministic-p search-node)
              (list traversal-type coord-list)))
        (optimal-path-segments-within-corridor
          (if (null (path-segment-deterministic-p search-node))
              (generate-optimal-path-segments-within-corridor search-node))
          (if (null (path-segment-deterministic-p search-node))
              (generate-optimal-path-segments-within-corridor search-node)))
        (optimal-path-segments
          (if (path-segment-deterministic-p search-node)
              (cons optimal-path-segment optimal-path-segments)
              (if (null (path-segment-deterministic-p search-node))
```

```
                    (if optimal-path-segments-within-corridor
                        (append optimal-path-segments-within-corridor
                          optimal-path-segments)
                        (setf pursuit-flag nil))))
              (if (path-segment-deterministic-p search-node)
                  (cons optimal-path-segment optimal-path-segments)
                  (if (null (path-segment-deterministic-p search-node))
                      (if optimal-path-segments-within-corridor
                          (append optimal-path-segments-within-corridor
                            optimal-path-segments)
                          (setf pursuit-flag nil))))))
          ((or (= (length optimization-search-node-list) 1) (null pursuit-flag))
           (reverse optimal-path-segments))))

;*****************************************************************************


(defun generate-optimal-path-segments-within-corridor (search-node)
  (let ((search-corridor
          (build-search-corridor search-node)))
    (if search-corridor
        (do* ((start-point
                (first (first search-corridor))
                (second (second (first optimal-path-segments))))
              (goal-point
                (first (last (first search-corridor))))
              (window-list
                (remove (first (first search-corridor))
                        (first search-corridor)) (rest window-list))
              (traversal-type-list
                (second search-corridor) (rest traversal-type-list))
              (permissible-headings-list
                (third search-corridor) (rest permissible-headings-list))
              (optimal-path-segments
                (list (generate-optimal-path-segment-within-corridor
                        start-point window-list traversal-type-list
                        permissible-headings-list))
                (cons (generate-optimal-path-segment-within-corridor
                        start-point window-list traversal-type-list
                        permissible-headings-list)
                      optimal-path-segments)))
             ((= (length window-list) 2)
              (cons (list (second traversal-type-list)
                          (list (second (second
                                  (first optimal-path-segments))) goal-point))
                    optimal-path-segments))))))

;*****************************************************************************


(defun generate-optimal-path-segment-within-corridor
       (start-point window-list traversal-type-list permissible-headings-list)
  (do* ((path-result
          (generate-path-segments-within-corridor
            start-point nil window-list traversal-type-list
             permissible-headings-list)
          (generate-path-segments-within-corridor
            start-point way-point-list pursuit-window-list traversal-type-list
```

```
   permissible-headings-list))
(path-cost
  (first path-result) (first path-result))
(path
  (second path-result) (second path-result))
(pursuit-edge-segments
  (third path-result) (third path-result))
(pursuit-edge-segment1
  (first pursuit-edge-segments) (first pursuit-edge-segments))
(pursuit-edge-segment2
  (second pursuit-edge-segments) (second pursuit-edge-segments))
(way-point-list
  (rest (second path-result)) (rest (second path-result)))
(path-result1
  (generate-path-segments-within-corridor
    start-point way-point-list
     (cons pursuit-edge-segment1
       (rest window-list)) traversal-type-list
        permissible-headings-list)
  (generate-path-segments-within-corridor
    start-point way-point-list
     (cons pursuit-edge-segment1
      (rest window-list)) traversal-type-list
       permissible-headings-list))
(path-cost1
  (first path-result1) (first path-result1))
(path1
  (second path-result1) (second path-result1))
(path-result2
  (generate-path-segments-within-corridor
    start-point way-point-list
     (cons pursuit-edge-segment2
      (rest window-list)) traversal-type-list
        permissible-headings-list)
  (generate-path-segments-within-corridor
    start-point way-point-list
     (cons pursuit-edge-segment2
      (rest window-list)) traversal-type-list
       permissible-headings-list))
(path-cost2
  (first path-result2) (first path-result2))
(path2
  (second path-result2) (second path-result2))
(pursuit-edge-segment-center
  (list (second (second path-result1))
        (second (second path-result2)))
  (list (second (second path-result1))
        (second (second path-result2))))
(pursuit-direction
  (if (< path-cost1 path-cost) '1
      (if (< path-cost2 path-cost) '2
          (if (and (>= path-cost1 path-cost)
                   (>= path-cost2 path-cost)) 'C)))
  (if (< path-cost1 path-cost) '1
      (if (< path-cost2 path-cost) '2
          (if (and (>= path-cost1 path-cost)
                   (>= path-cost2 path-cost)) 'C))))
(pursuit-direction1
  (if (equal pursuit-direction '1) t nil)
  (if (equal pursuit-direction '1) t nil))
```

```
            (pursuit-direction2
              (if (equal pursuit-direction '2) t nil)
              (if (equal pursuit-direction '2) t nil))
            (pursuit-direction-center
              (if (equal pursuit-direction 'C) t nil)
              (if (equal pursuit-direction 'C) t nil))
            (pursuit-window-list
              (if pursuit-direction1
                  (cons pursuit-edge-segment1 (rest window-list))
                  (if pursuit-direction2
                      (cons pursuit-edge-segment2 (rest window-list))
                      (if pursuit-direction-center
                          (cons pursuit-edge-segment-center
                                (rest window-list)))))
              (if pursuit-direction1
                  (cons pursuit-edge-segment1 (rest window-list))
                  (if pursuit-direction2
                      (cons pursuit-edge-segment2 (rest window-list))
                      (if pursuit-direction-center
                          (cons pursuit-edge-segment-center
                                (rest window-list))))))
            (threshold 0.001)
            (tolerance 0.001)
            (optimization-edge-distance
              (if pursuit-direction1
                  (distance-point-to-point (first (second path))
                                           (second (second path))
                                           (first (second path1))
                                           (second (second path1)))
                  (distance-point-to-point (first (second path))
                                           (second (second path))
                                           (first (second path2))
                                           (second (second path2))))
              (if pursuit-direction1
                  (distance-point-to-point (first (second path))
                                           (second (second path))
                                           (first (second path1))
                                           (second (second path1)))
                  (distance-point-to-point (first (second path))
                                           (second (second path))
                                           (first (second path2))
                                           (second (second path2))))))
          ((or (< optimization-edge-distance threshold)
               (equal-within-tolerance path-cost path-cost1 tolerance)
               (equal-within-tolerance path-cost path-cost2 tolerance))
           (if pursuit-direction1
               (list (first traversal-type-list)
               (list (first path1) (second path1)))
               (if pursuit-direction2
                   (list (first traversal-type-list)
                   (list (first path2) (second path2)))
                   (if pursuit-direction-center
                       (list (first traversal-type-list)
                       (list (first path) (second path)))))))))

(defun generate-path-segments-within-corridor
       (start-point way-point-list window-list traversal-type-list
        permissible-headings-list)
  (do* ((optimization-window-list
         window-list
```

```
  (rest optimization-window-list))
 (optimization-traversal-type-list
  traversal-type-list
  (rest optimization-traversal-type-list))
 (optimization-permissible-headings-list
  permissible-headings-list
  (rest optimization-permissible-headings-list))
 (deterministic-approach-heading
  (if
    (= (length (first (first optimization-permissible-headings-list)))
       1) t nil)
  (if
    (= (length (first (first optimization-permissible-headings-list)))
       1) t nil))
 (optimization-way-point-list
  way-point-list
  (rest optimization-way-point-list))
 (optimization-start-point start-point way-point)
 (way-point
  (if
    deterministic-approach-heading
    (generate-way-point-deterministic
      start-point (first (first optimization-window-list))
       (second (first optimization-window-list))
         (first optimization-permissible-headings-list))
    (if
      optimization-way-point-list
      (first optimization-way-point-list)
        (generate-way-point-non-deterministic
          (first (first optimization-window-list))
            (second (first optimization-window-list)))))
  (if
    deterministic-approach-heading
    (generate-way-point-deterministic
      start-point (first (first optimization-window-list))
       (second (first optimization-window-list))
         (first optimization-permissible-headings-list))
    (if
      optimization-way-point-list
      (first optimization-way-point-list)
        (generate-way-point-non-deterministic
          (first (first optimization-window-list))
            (second (first optimization-window-list))))))
 (optimization-edge-segments
  (list (list way-point (first (first window-list)))
  (list way-point (second (first window-list)))))
 (path-segments
  (list (list (first optimization-traversal-type-list)
              (list optimization-start-point way-point)))
  (cons (list (first optimization-traversal-type-list)
              (list optimization-start-point way-point))
        path-segments)))
((= (length (second optimization-window-list)) 3)
 (let ((path-list
        (synthesize-optimal-path-segments
          (reverse
            (cons (list (second traversal-type-list)
                        (list way-point
                              (second optimization-window-list)))
                  path-segments)))))
```

```
          (list (first path-list) (second path-list)
                                  optimization-edge-segments)))))


(defun synthesize-optimal-path-segments (optimal-path-segments)
  (let ((optimal-path-segment-costs nil)
        (optimal-path nil)
        (optimal-path-cost nil))
    (dolist (optimal-path-segment optimal-path-segments)
      (let* ((traversal-type (first optimal-path-segment))
             (coord-list (second optimal-path-segment))
             (x1 (first (first coord-list)))
             (y1 (second (first coord-list)))
             (z1 (third (first coord-list)))
             (x2 (first (second coord-list)))
             (y2 (second (second coord-list)))
             (z2 (third (second coord-list))))
        (if (or (equal traversal-type 'I)
                (equal traversal-type 'II))
            (setf optimal-path-segment-costs
                  (cons (* (distance-point-to-point x1 y1 x2 y2)
                           *optimal-cost-rate*)
                        optimal-path-segment-costs))
            (if (equal traversal-type 'IV)
                (setf optimal-path-segment-costs
                      (cons (abs (- z2 z1)) optimal-path-segment-costs))))))
    (setf optimal-path-cost (apply '+ optimal-path-segment-costs))
    (setf optimal-path
          (append (mapcar 'first (mapcar 'second optimal-path-segments))
                  (list (second (second (first
                                  (last optimal-path-segments)))))))
    (list optimal-path-cost optimal-path)))


;*****************************************************************************
;
; Path Planning Model: Miscellaneous Path Planning Support Functions
;
;*****************************************************************************


(defun build-search-corridor (search-node)
  (let* ((start-point
           (get-xyz-coord-from-vertex
             (first (search-node-window (eval search-node)))))
         (goal-point
           (get-xyz-coord-from-vertex
             (first (last (search-node-window (eval search-node))))))
         (window-list (rest (search-node-window (eval search-node))))
         (window-coord-lists
           (mapcar '(lambda (window-coord-list)
             (mapcar 'get-xyz-coord-from-vertex window-coord-list))
               (mapcar 'edge-vertex-list
                 (mapcar 'eval
                   (remove (first (last window-list)) window-list)))))
         (permissible-headings-list
           (search-node-permissible-headings (eval search-node)))
         (reversed-permissible-headings-list
           (mapcar '(lambda (permissible-headings)
                    (if (= (length (first permissible-headings)) 1)
```

```
                (list
                  (list
                    (list
                      (reverse-heading
                        (first (first (first permissible-headings))))
                      'cl)))
                (list
                  (list
                    (list (reverse-heading
                      (first (first (first permissible-headings))))
                        (second (first (first permissible-headings))))
                      (list (reverse-heading
                        (first (second (first permissible-headings))))
                          (second (second (first
                            permissible-headings))))))))
            permissible-headings-list))
(traversal-type-list (search-node-traversal-type (eval search-node)))
(optimization-coord-lists
  (generate-optimization-windows
    start-point goal-point window-coord-lists
     permissible-headings-list))
(revised-optimization-coord-lists
  (if optimization-coord-lists
      (generate-optimization-windows
        (first optimization-coord-lists)
          (first (last optimization-coord-lists))
            (mapcar 'reverse (remove (first optimization-coord-lists)
                                     (remove (first
                                               (last optimization-coord-lists))
                                       optimization-coord-lists)))
(reverse reversed-permissible-headings-list)))))
  (if revised-optimization-coord-lists
      (let* ((start-point (first revised-optimization-coord-lists))
             (goal-point (first (last
               revised-optimization-coord-lists)))
             (intermediate-edges
               (remove start-point
                 (remove goal-point
                   revised-optimization-coord-lists))))
        (list
          (cons
            (list
              (make-significant-figures (first start-point) 2)
              (make-significant-figures (second start-point) 2)
              (make-significant-figures (third start-point) 2))
            (append (mapcar '(lambda (edge)
              (mapcar '(lambda (point)
                (list
                  (make-significant-figures (first point) 2)
                  (make-significant-figures (second point) 2)
                  (make-significant-figures (third point) 2)))
                  edge))
                intermediate-edges)
              (list (list
                    (make-significant-figures (first goal-point) 2)
                    (make-significant-figures (second goal-point) 2)
                    (make-significant-figures (third goal-point) 2)))))
          traversal-type-list
          (mapcar '(lambda (permissible-headings)
                   (mapcar '(lambda (permissible-heading-list)
```

```
                                        (list (make-significant-figures
                                                (first permissible-heading-list) 2)
                                              (second permissible-heading-list)))
                                        (first permissible-headings)))
                                permissible-headings-list))))))


(defun generate-optimization-windows
       (start-point goal-point window-coord-lists permissible-headings-list)
  (do* ((optimization-permissible-headings-list
          permissible-headings-list
          (rest optimization-permissible-headings-list))
        (optimization-permissible-headings
          (first optimization-permissible-headings-list)
          (first optimization-permissible-headings-list))
        (deterministic-approach-heading
          (if (= (length (first optimization-permissible-headings)) 1) t nil)
          (if (= (length (first optimization-permissible-headings)) 1) t nil))
        (optimization-window-coord-lists
          window-coord-lists
          (rest optimization-window-coord-lists))
        (point1
          (first (first optimization-window-coord-lists))
          (first (first optimization-window-coord-lists)))
        (point2
          (second (first optimization-window-coord-lists))
          (second (first optimization-window-coord-lists)))
        (pursuit-flag t pursuit-flag)
        (way-point1
          (generate-way-point-deterministic
            start-point point1 point2
            (list (list (first (first optimization-permissible-headings)))))
          (if (or (point-equal-p
                    (first optimization-point1) (second optimization-point1)
                    (first point1) (second point1))
                  (point-equal-p
                    (first optimization-point1) (second optimization-point1)
                    (first point2) (second point2)))
              optimization-point1
              (if deterministic-approach-heading
                  (generate-way-point-deterministic
                    optimization-point1 point1 point2
                    optimization-permissible-headings)
                  (generate-way-point-deterministic
                    optimization-point1 point1 point2
                    (list
                      (list
                        (first
                          (first optimization-permissible-headings)))))))))
        (way-point2
          (generate-way-point-deterministic
            start-point point1 point2
            (list (list (second (first optimization-permissible-headings)))))
          (if (or (point-equal-p
                    (first optimization-point2) (second optimization-point2)
                    (first point1) (second point1))
                  (point-equal-p
                    (first optimization-point2) (second optimization-point2)
                    (first point2) (second point2)))
              optimization-point2
```

```
      (if deterministic-approach-heading
          (generate-way-point-deterministic
            optimization-point2 point1 point2
            optimization-permissible-headings)
          (generate-way-point-deterministic
            optimization-point2 point1 point2
            (list
              (list
                (second
                  (first optimization-permissible-headings)))))))))
(optimization-coord-lists
  (list
    (let ((point-in-range
            (point-in-range-simple-p
              point1 start-point
               (first (first (first optimization-permissible-headings)))
                 (first (second (first
                   optimization-permissible-headings))))))
      (if
        (and (null way-point1) (null way-point2) point-in-range)
        (list point1 point2)
        (if
          (and (null way-point1) (null way-point2)
                                 (null point-in-range))
          (setf pursuit-flag nil)
          (if
            (and (null way-point1) point-in-range)
            (list point1 way-point2)
            (if
              (and (null way-point1) (null point-in-range))
              (list point2 way-point2)
              (if
                (and (null way-point2) point-in-range)
                (list way-point1 point1)
                (if
                  (and (null way-point2) (null point-in-range))
                  (list way-point1 point2)
                  (list way-point1 way-point2)))))))))
  (cons
    (if
      deterministic-approach-heading
      (let ((point-in-corridor
              (point-in-corridor-p
                point1 optimization-point1 optimization-point2
                 (first (first (first
                   optimization-permissible-headings))))))
        (if
          (and (null way-point1) (null way-point2) point-in-corridor)
          (list point1 point2)
          (if
            (and (null way-point1) (null way-point2)
                                   (null point-in-corridor))
            (setf pursuit-flag nil))
            (if
              (and (null way-point1) point-in-corridor)
              (list point1 way-point2)
              (if
                (and (null way-point1) (null point-in-corridor))
                (list point2 way-point2)
                (if
```

```
                                (and (null way-point2) point-in-corridor)
                                (list way-point1 point1)
                                (if
                                  (and (null way-point2) (null point-in-corridor))
                                  (list way-point1 point2)
                                  (list way-point1 way-point2)))))))
                    (if (null deterministic-approach-heading)
                        (let ((point-in-range
                                (point-in-range-p
                                  point1 optimization-point1 optimization-point2
                                   (first (first (first
                                     optimization-permissible-headings)))
                                      (first (second (first
                                        optimization-permissible-headings))))))
                              (if
                                (and (null way-point1) (null way-point2) point-in-range)
                                (list point1 point2)
                                (if
                                  (and (null way-point1) (null way-point2)
                                                         (null point-in-range))
                                  (setf pursuit-flag nil)
                                  (if
                                    (and (null way-point1) point-in-range)
                                    (list point1 way-point2)
                                    (if
                                      (and (null way-point1) (null point-in-range))
                                      (list point2 way-point2)
                                      (if
                                        (and (null way-point2) point-in-range)
                                        (list way-point1 point1)
                                        (if
                                          (and (null way-point2) (null point-in-range))
                                          (list way-point1 point2)
                                          (list way-point1 way-point2)))))))))
                            optimization-coord-lists))
                         (optimization-point1
                           (first (first optimization-coord-lists))
                           (first (first optimization-coord-lists)))
                         (optimization-point2
                           (second (first optimization-coord-lists))
                           (second (first optimization-coord-lists))))
                        ((or (= (length optimization-window-coord-lists) 1) (null pursuit-flag))
                         (if (null pursuit-flag) nil
                             (cons goal-point
                                   (append optimization-coord-lists (list start-point)))))))


(defun generate-way-point
       (start-point optimization-edge-segment permissible-headings)
  (if (= (length (first permissible-headings)) 1)
      (generate-way-point-deterministic start-point
                                        (first optimization-edge-segment)
                                        (second optimization-edge-segment)
                                        permissible-headings)
      (generate-way-point-non-deterministic (first optimization-edge-segment)
                                            (second optimization-edge-segment))))


(defun generate-way-point-deterministic
       (start-point edge-point1 edge-point2 permissible-headings)
```

```
  (let* ((start-pt-x (first start-point))
         (start-pt-y (second start-point))
         (edge-pt1-x (first edge-point1))
         (edge-pt1-y (second edge-point1))
         (edge-pt2-x (first edge-point2))
         (edge-pt2-y (second edge-point2))
         (heading-pt
         (heading-point start-pt-x start-pt-y
                        (first (first (first permissible-headings)))))
         (heading-pt-x (first heading-pt))
         (heading-pt-y (second heading-pt))
         (intersect-pt
           (intersect start-pt-x start-pt-y heading-pt-x heading-pt-y
                      edge-pt1-x edge-pt1-y edge-pt2-x edge-pt2-y))
         (intersect-pt-x (if intersect-pt (first intersect-pt)))
         (intersect-pt-y (if intersect-pt (second intersect-pt))))
    (if intersect-pt
        (list (first intersect-pt)
              (second intersect-pt)
              (get-elevation-from-edge-point
                (list intersect-pt-x intersect-pt-y)
                      edge-point1 edge-point2)))))


(defun generate-way-point-non-deterministic (edge-point1 edge-point2)
  (let*((edge-pt1-x (first edge-point1))
        (edge-pt1-y (second edge-point1))
        (edge-pt2-x (first edge-point2))
        (edge-pt2-y (second edge-point2))
        (edge-midpoint
          (line-midpoint edge-pt1-x edge-pt1-y edge-pt2-x edge-pt2-y))
        (edge-midpoint-x (first edge-midpoint))
        (edge-midpoint-y (second edge-midpoint))
        (edge-midpoint-z
          (get-elevation-from-edge-point
            (list edge-midpoint-x edge-midpoint-y) edge-point1 edge-point2)))
    (list edge-midpoint-x edge-midpoint-y edge-midpoint-z)))


(defun path-segment-deterministic-p (search-node)
  (if (remove-if-not 'null
                     (mapcar 'vertex-p
                             (mapcar 'eval
                                     (search-node-window
                                       (eval search-node))))) nil t))


(defun path-segment-non-deterministic-p (search-node)
  (if (path-segment-deterministic-p search-node) nil t))


(defun pursuit-edge-segment-point-equal-p (pursuit-edge-segment-pt1
                                           pursuit-edge-segment-pt2)
  (let ((tolerance 0.001))
    (if (and (equal-within-tolerance
               (first pursuit-edge-segment-pt1)
               (first pursuit-edge-segment-pt2) tolerance)
             (equal-within-tolerance
               (second pursuit-edge-segment-pt1)
               (second pursuit-edge-segment-pt2) tolerance)
```

```
                (equal-within-tolerance
                (third pursuit-edge-segment-pt1)
                (third pursuit-edge-segment-pt2) tolerance))
            t nil)))


(defun point-in-range-p (target-point point1 point2 heading1 heading2)
  (let* ((point1-x (first point1))
         (point1-y (second point1))
         (point2-x (first point2))
         (point2-y (second point2))
         (target-point-x (first target-point))
         (target-point-y (second target-point))
         (test-heading1
           (heading point1-x point1-y target-point-x target-point-y))
         (test-heading2
           (heading point2-x point2-y target-point-x target-point-y)))
    (if (or (heading-range-p test-heading1 heading1 heading2)
            (heading-range-p test-heading2 heading1 heading2)) t
        (let* ((heading-pt11 (heading-point point1-x point1-y heading1))
               (heading-pt11-x (first heading-pt11))
               (heading-pt11-y (second heading-pt11))
               (heading-pt12 (heading-point point1-x point1-y heading2))
               (heading-pt12-x (first heading-pt12))
               (heading-pt12-y (second heading-pt12))
               (heading-pt21 (heading-point point2-x point2-y heading1))
               (heading-pt21-x (first heading-pt21))
               (heading-pt21-y (second heading-pt21))
               (heading-pt22 (heading-point point2-x point2-y heading2))
               (heading-pt22-x (first heading-pt22))
               (heading-pt22-y (second heading-pt22))
               (line-equation11
                 (line-equation point1-x point1-y
                   heading-pt11-x heading-pt11-y))
               (line-equation12
                 (line-equation point1-x point1-y
                   heading-pt12-x heading-pt12-y))
               (line-equation21
                 (line-equation point2-x point2-y
                   heading-pt21-x heading-pt21-y))
               (line-equation22
                 (line-equation point2-x point2-y
                   heading-pt22-x heading-pt22-y))
               (le-solution11
                 (line-equation-solution line-equation11
                   target-point-x target-point-y))
               (le-solution12
                 (line-equation-solution line-equation12
                   target-point-x target-point-y))
               (le-solution21
                 (line-equation-solution line-equation21
                   target-point-x target-point-y))
               (le-solution22
                 (line-equation-solution line-equation22
                   target-point-x target-point-y)))
          (if (or (and (plusp le-solution11)
                       (minusp le-solution21))
                  (and (minusp le-solution11)
                       (plusp le-solution21))
                  (and (plusp le-solution12)
```

```
                    (minusp le-solution22))
               (and (minusp le-solution12)
                    (plusp le-solution22))) t)))))


(defun point-in-range-simple-p (target-point point heading1 heading2)
  (let* ((point-x (first point))
         (point-y (second point))
         (target-point-x (first target-point))
         (target-point-y (second target-point))
         (test-heading (heading point-x point-y target-point-x target-point-y)))
    (if (heading-range-p test-heading heading1 heading2) t)))


(defun point-in-corridor-p (target-point point1 point2 heading)
  (let* ((point1-x (first point1))
         (point1-y (second point1))
         (point2-x (first point2))
         (point2-y (second point2))
         (target-point-x (first target-point))
         (target-point-y (second target-point))
         (heading-pt11 (heading-point point1-x point1-y heading))
         (heading-pt11-x (first heading-pt11))
         (heading-pt11-y (second heading-pt11))
         (heading-pt21 (heading-point point2-x point2-y heading))
         (heading-pt21-x (first heading-pt21))
         (heading-pt21-y (second heading-pt21))
         (line-equation11
           (line-equation point1-x point1-y heading-pt11-x heading-pt11-y))
         (line-equation21
           (line-equation point2-x point2-y heading-pt21-x heading-pt21-y))
         (le-solution11
           (line-equation-solution line-equation11 target-point-x
                                                   target-point-y))
         (le-solution21
           (line-equation-solution line-equation21 target-point-x
                                                   target-point-y)))
    (if (or (and (plusp le-solution11)
                 (minusp le-solution21))
            (and (minusp le-solution11)
                 (plusp le-solution21))) t)))


(defun least-cost-p (search-node1 search-node2)
  (let* ((cost-from-start1
           (first (search-node-cost-from-start (eval search-node1))))
         (cost-from-start2
           (first (search-node-cost-from-start (eval search-node2))))
         (estimate-to-goal1
           (first (search-node-estimate-to-goal (eval search-node1))))
         (estimate-to-goal2
           (first (search-node-estimate-to-goal (eval search-node2))))
         (path-cost1 (+ cost-from-start1 estimate-to-goal1))
           (path-cost2 (+ cost-from-start2 estimate-to-goal2)))
    (if (< path-cost1 path-cost2) t nil)))


(defun path-length (path)
  (cond ((null (rest path)) 0.0)
        (t (let* ((window1 (eval (first path)))
```

```
                (window2 (eval (first (rest path))))
                (x1 (vertex-x-coord window1))
                (y1 (vertex-y-coord window1))
                (x2 (vertex-x-coord window2))
                (y2 (vertex-y-coord window2)))
            (+ (distance-point-to-point x1 y1 x2 y2)
               (path-length (rest path)))))))


;*****************************************************************************
;
; Path-Planning Model: Definition and Initialization of Global Variables
;
;*****************************************************************************


(defvar *agenda-length*)
(defvar *local-optimal-path-cost*)
(defvar *local-optimal-path*)
(defvar *total-local-optimal-paths*)
(defvar *global-optimal-path-cost*)
(defvar *global-optimal-path*)


(setf *agenda-length* nil)
(setf *local-optimal-path-cost* nil)
(setf *local-optimal-path* nil)
(setf *total-local-optimal-paths* 0)
(setf *global-optimal-path-cost* nil)
(setf *global-optimal-path* nil)


;*****************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-COMMAND-CONTROL-UTILITIES
;
; Functions: MINIMUM-ENERGY-PATH-PLANNING ()
;            MPP-INITIALIZE ()
;            MPP-INITIALIZE-WITH-CURRENT-MAP ()
;            ASSERT-MAP (filename)
;            ASSERT-VEHICLE (vehicle)
;            ASSERT-REGION-TOTALS ()
;            ASSERT-START ()
;            ASSERT-START-COORDINATES (xcoord ycoord)
;            ASSERT-GOAL ()
;            ASSERT-GOAL-COORDINATES (xcoord ycoord)
;            ASSERT-START-GOAL-VISIBILITY ()
;            RETRACT-START-GOAL-VISIBILITY ()
;            ASSERT-REGION ()
;            MISSION-GO ()
;
; Global Variables: *start-location*
;                   *goal-location*
;                   *critical-coasting-angle*
;                   *critical-braking-angle*
;                   *critical-stability-angle*
;                   *optimal-cost-rate*
;                   *current-map*
;                   *current-region*
;                   *current-region-examined*
;                   *current-vehicle*
;                   *motion-resistance-lower*
;                   *motion-resistance-upper*
;                   *total-isotropic-regions*
;                   *total-anisotropic-safe-regions*
;                   *total-anisotropic-partially-safe-regions*
;                   *total-obstacle-regions*
;                   *termination-flag*
;                   *global-planning-time*
;
;*****************************************************************************


;*****************************************************************************


(defun minimum-energy-path-planning ()
  (mpp-initialize)
  (set-display)
  (do ((termination-flag nil *termination-flag*))
      ((equal termination-flag t) t)
    (mpp-top-level-menu-window))
  (kill-display))


(defun mpp-initialize ()
  (load-vehicles "VEHICLE-PROFILES")
  (setf *start-location* nil)
  (setf *goal-location* nil)
  (setf *critical-coasting-angle* nil)
  (setf *critical-braking-angle* nil)
  (setf *critical-stability-angle* nil)
```

```
  (setf *optimal-cost-rate* nil)
  (setf *current-map* nil)
  (setf *current-region* nil)
  (setf *current-region-examined* nil)
  (setf *current-region-examined-coords* nil)
  (setf *current-vehicle* nil)
  (setf *motion-resistance-lower* nil)
  (setf *motion-resistance-upper* nil)
  (setf *local-optimal-path-cost* nil)
  (setf *local-optimal-path* nil)
  (setf *total-local-optimal-paths* 0)
  (setf *agenda-length* 0)
  (setf *global-optimal-path-cost* nil)
  (setf *global-optimal-path* nil)
  (setf *global-planning-time* nil)
  (setf *total-isotropic-regions* 0)
  (setf *total-anisotropic-safe-regions* 0)
  (setf *total-anisotropic-partially-safe-regions* 0)
  (setf *total-obstacle-regions* 0)
  (setf *termination-flag* nil) t)


(defun mpp-initialize-with-current-map ()
  (load-vehicles "VEHICLE-PROFILES")
  (setf *start-location* nil)
  (setf *goal-location* nil)
  (setf *critical-coasting-angle* nil)
  (setf *critical-braking-angle* nil)
  (setf *critical-stability-angle* nil)
  (setf *optimal-cost-rate* nil)
  (setf *current-region* nil)
  (setf *current-region-examined* nil)
  (setf *current-region-examined-coords* nil)
  (setf *current-vehicle* nil)
  (setf *motion-resistance-lower* nil)
  (setf *motion-resistance-upper* nil)
  (setf *local-optimal-path-cost* nil)
  (setf *local-optimal-path* nil)
  (setf *total-local-optimal-paths* 0)
  (setf *global-optimal-path-cost* nil)
  (setf *global-optimal-path* nil)
  (setf *global-planning-time* nil)
  (setf *total-isotropic-regions* (length *background-region-list*))
  (setf *total-anisotropic-safe-regions* 0)
  (setf *total-anisotropic-partially-safe-regions* 0)
  (setf *total-obstacle-regions* 0)
  (setf *termination-flag* nil) t)


(defun assert-map (filename)
  (setf *current-map* filename)
  (load-map filename) t)


(defun assert-vehicle (vehicle)
  (setf *current-vehicle* vehicle)
  (setf *critical-coasting-angle* (vehicle-coasting-slope (eval vehicle)))
  (setf *critical-braking-angle* (vehicle-gradient-slope (eval vehicle)))
  (setf *critical-stability-angle* (vehicle-contour-slope (eval vehicle)))
  (setf *motion-resistance-lower*
```

```
        (make-significant-figures
          (tan (degrees-to-radians *critical-coasting-angle*)) 3))
  (setf *motion-resistance-upper*
        (make-significant-figures
          (tan (degrees-to-radians *critical-braking-angle*)) 3))
  (setf *optimal-cost-rate* *motion-resistance-lower*)
  (build-braking-constraints)
  (build-region-type) t)


(defun assert-region-totals ()
  (setf *total-isotropic-regions* 0)
  (setf *total-anisotropic-safe-regions* 0)
  (setf *total-anisotropic-partially-safe-regions* 0)
  (setf *total-obstacle-regions* 0)
  (dolist (region (append *region-list* *background-region-list*))
    (if (isotropic-region-p region)
        (setf *total-isotropic-regions* (1+ *total-isotropic-regions*))
        (if (anisotropic-safe-region-p region)
            (setf *total-anisotropic-safe-regions*
                  (1+ *total-anisotropic-safe-regions*))
            (if (anisotropic-partially-safe-region-p region)
                (setf *total-anisotropic-partially-safe-regions*
                   (1+ *total-anisotropic-partially-safe-regions*))
                (if (obstacle-region-p region)
                    (setf *total-obstacle-regions*
                          (1+ *total-obstacle-regions*)))))))) t)


(defun assert-start ()
  (let* ((coordinate-list (get-mouse-coordinates))
         (xcoord (first coordinate-list))
         (ycoord (second coordinate-list))
         (region-list (get-region-from-point xcoord ycoord))
         (start-region (if (= (length region-list) 1) (first region-list))))
    (if start-region
        (let ((zcoord (get-elevation-from-interior-point
                        (list xcoord ycoord) start-region)))
          (setf s-v (build-vertex
                      xcoord ycoord zcoord nil
                      (get-vertices-and-edges-from-region start-region)))
          (setf *start-location* (list xcoord ycoord zcoord))
          (setf *current-region* start-region) t)
     (setf s-v nil))))


(defun assert-start-coordinates (xcoord ycoord)
  (let* ((region-list (get-region-from-point xcoord ycoord))
         (start-region (if (= (length region-list) 1) (first region-list))))
    (if start-region
        (let ((zcoord
                (get-elevation-from-interior-point
                  (list xcoord ycoord) start-region)))
          (setf s-v (build-vertex
                      xcoord ycoord zcoord nil
                       (get-vertices-and-edges-from-region start-region)))
          (setf *start-location* (list xcoord ycoord zcoord))
          (setf *current-region* start-region) t)
        (setf s-v nil))))
```

```
(defun assert-goal ()
  (let* ((coordinate-list (get-mouse-coordinates))
         (xcoord (first coordinate-list))
         (ycoord (second coordinate-list))
         (region-list (get-region-from-point xcoord ycoord))
         (goal-region (if (= (length region-list) 1) (first region-list))))
    (if goal-region
        (let ((zcoord
                (get-elevation-from-interior-point
                  (list xcoord ycoord) goal-region)))
          (setf g-v (build-vertex
                      xcoord ycoord zcoord nil
                       (get-vertices-and-edges-from-region goal-region)))
          (setf *goal-location* (list xcoord ycoord zcoord)) t)
        (setf g-v nil))))


(defun assert-goal-coordinates (xcoord ycoord)
  (let* ((region-list (get-region-from-point xcoord ycoord))
         (goal-region (if (= (length region-list) 1) (first region-list))))
    (if goal-region
        (let ((zcoord
                (get-elevation-from-interior-point
                  (list xcoord ycoord) goal-region)))
          (setf g-v (build-vertex
                      xcoord ycoord zcoord nil
                       (get-vertices-and-edges-from-region goal-region)))
          (setf *goal-location* (list xcoord ycoord zcoord)) t)
        (setf g-v nil))))


(defun assert-start-goal-visibility ()
  (if (or (null (eval 's-v))
          (null (eval 'g-v))) nil
      (let ()
        (dolist (window (vertex-visibility-list (eval 's-v)))
          (if (edge-p (eval window))
              (setf (edge-visibility-list (eval window))
                    (cons 's-v (edge-visibility-list (eval window))))
              (if (vertex-p (eval window))
                  (setf (vertex-visibility-list (eval window))
                        (cons 's-v (vertex-visibility-list (eval window)))))))
        (dolist (window (vertex-visibility-list (eval 'g-v)))
          (if (edge-p (eval window))
              (setf (edge-visibility-list (eval window))
                    (cons 'g-v (edge-visibility-list (eval window))))
              (if (vertex-p (eval window))
                  (setf (vertex-visibility-list (eval window))
                        (cons 'g-v (vertex-visibility-list (eval window)))))))
        (if (equal (get-region-from-point (vertex-x-coord (eval 's-v))
                                          (vertex-y-coord (eval 's-v)))
                   (get-region-from-point (vertex-x-coord (eval 'g-v))
                                          (vertex-y-coord (eval 'g-v))))
            (let ()
              (setf (vertex-visibility-list (eval 's-v))
                    (cons 'g-v (vertex-visibility-list (eval 's-v))))
              (setf (vertex-visibility-list (eval 'g-v))
                    (cons 's-v (vertex-visibility-list (eval 'g-v))))))) t)))
```

```
(defun retract-start-goal-visibility ()
  (if (or (null (eval 's-v))
          (null (eval 'g-v))) nil
      (let* ((start-window-visibility-list (vertex-visibility-list (eval 's-v)))
             (goal-window-visibility-list (vertex-visibility-list (eval 'g-v))))
        (dolist (start-window-visible start-window-visibility-list)
          (if (vertex-p (eval start-window-visible))
              (setf (vertex-visibility-list (eval start-window-visible))
                    (remove 's-v (vertex-visibility-list
                      (eval start-window-visible))))
              (setf (edge-visibility-list (eval start-window-visible))
                    (remove 's-v (edge-visibility-list
                      (eval start-window-visible))))))
        (dolist (goal-window-visible goal-window-visibility-list)
          (if (vertex-p (eval goal-window-visible))
              (setf (vertex-visibility-list (eval goal-window-visible))
                    (remove 'g-v (vertex-visibility-list
                      (eval goal-window-visible))))
              (setf (edge-visibility-list (eval goal-window-visible))
                    (remove 'g-v (edge-visibility-list
                      (eval goal-window-visible)))))) t)))


(defun assert-region ()
  (let* ((coordinate-list (get-mouse-coordinates))
         (xcoord (first coordinate-list))
         (ycoord (second coordinate-list))
         (region-list (get-region-from-point xcoord ycoord))
         (region (if (= (length region-list) 1) (first region-list))))
    (if region
        (let ()
          (setf *current-region-examined* region)
          (setf *current-region-examined-coords* (list xcoord ycoord))))) t)


(defun mission-go ()
  (if (or (null *start-location*)
          (null *goal-location*)
          (null *current-vehicle*)) nil t))


;*****************************************************************************


(defvar *start-location*)
(defvar *goal-location*)
(defvar *critical-coasting-angle*)
(defvar *critical-braking-angle*)
(defvar *critical-stability-angle*)
(defvar *optimal-cost-rate*)
(defvar *current-map*)
(defvar *current-region*)
(defvar *current-region-examined*)
(defvar *current-region-examined-coords*)
(defvar *current-vehicle*)
(defvar *motion-resistance-lower*)
(defvar *motion-resistance-upper*)
(defvar *total-isotropic-regions*)
(defvar *total-anisotropic-safe-regions*)
(defvar *total-anisotropic-partially-safe-regions*)
```

```
(defvar *total-obstacle-regions*)
(defvar *termination-flag*)
(defvar *global-planning-time*)


(setf *start-location* nil)
(setf *goal-location* nil)
(setf *critical-coasting-angle* nil)
(setf *critical-braking-angle* nil)
(setf *critical-stability-angle* nil)
(setf *optimal-cost-rate* nil)
(setf *current-map* nil)
(setf *current-region* nil)
(setf *current-region-examined* nil)
(setf *current-region-examined-coords* nil)
(setf *current-vehicle* nil)
(setf *motion-resistance-lower* nil)
(setf *motion-resistance-upper* nil)
(setf *total-isotropic-regions* (length *background-region-list*))
(setf *total-anisotropic-safe-regions* 0)
(setf *total-anisotropic-partially-safe-regions* 0)
(setf *total-obstacle-regions* 0)
(setf *termination-flag* nil)
(setf *global-planning-time* nil)


;*******************************************************************************
```

```
;*****************************************************************************
;
; File: MPP-DISPLAY-MAP-UTILITIES
;
; Flavors: SPECIAL-WINDOW ()
;
; Functions: CREATE-TERRAIN-WINDOW ()
;            CREATE-PROFILE-WINDOW ()
;            CREATE-TOP-LEVEL-MENU-WINDOW ()
;            CREATE-MAP-MENU-WINDOW ()
;            CREATE-VEHICLE-MENU-WINDOW ()
;            CREATE-MISSION-MENU-WINDOW ()
;            INITIALIZE-TERRAIN-WINDOW ()
;            INITIALIZE-PROFILE-WINDOW ()
;            SET-DISPLAY ()
;            RESET-DISPLAY ()
;            REFRESH-DISPLAY ()
;            REFRESH-DISPLAY-WITH-LOCAL-PATH
;            REFRESH-DISPLAY-WITH-GLOBAL-PATH
;            KILL-DISPLAY ()
;            DELAY-LOOP ()
;            SELECTION-MAP ()
;            SELECTION-VEHICLE ()
;            SELECTION-MISSION ()
;            SELECTION-REGION ()
;            SELECTION-PLAN ()
;            SELECTION-RESET ()
;            SELECTION-QUIT ()
;            SELECTION-MAP-LEVEL-1 ()
;            SELECTION-VEHICLE-LEVEL-1 ()
;            SELECTION-MISSION-START ()
;            SELECTION-MISSION-GOAL ()
;            MPP-TOP-LEVEL-MENU-WINDOW ()
;            DISPLAY-MAP ()
;            DISPLAY-MAP-SUMMARY ()
;            DISPLAY-VEHICLE ()
;            DISPLAY-MISSION ()
;            DISPLAY-REGION ()
;            DISPLAY-STATISTICS ()
;            DRAW-EDGE (edge)
;            DRAW-THICK-EDGE (edge)
;            DRAW-PATH-SEGMENT (coord-list1 coord-list2)
;            DRAW-PATH (path-list)
;            GET-MOUSE-COORDINATES ()
;            SPECIAL-EFFECTS ()
;
; Global Variables: *terrain-window*
;                   *profile-window*
;                   *top-level-menu-window*
;                   *map-menu-window*
;                   *vehicle-menu-window*
;                   *mission-menu-window*
;
;*****************************************************************************
```

```
;*****************************************************************************

(defflavor special-window ()
           (tv:margin-space-mixin
            tv:window))


(defun create-terrain-window ()
  (setf *terrain-window*
        (tv:make-window 'special-window
                        :margin-space 1
                        :position '(64 43)
                        :width 640
                        :height 748
                        :name "TERRAIN DISPLAY"
                        :borders 10
                        :blinker-p nil
                        :expose-p t
                        :save-bits t)) t)


(defun create-profile-window ()
  (setf *profile-window*
        (tv:make-window 'special-window
                        :margin-space 1
                        :position '(704 43)
                        :width 420
                        :height 748
                        :name "PROFILE DISPLAY"
                        :borders 10
                        :blinker-p nil
                        :expose-p t
                        :save-bits t)) t)


(defun create-top-level-menu-window ()
  (setf *top-level-menu-window*
        (tv: make-window
          'tv:pop-up-menu
          ':label "Minimum-energy Path Planning"
          ':borders 3
          ':item-list '(("Assert Map" :funcall selection-map)
                        ("Assert Vehicle" :funcall selection-vehicle)
                        ("Assert Mission" :funcall selection-mission)
                        ("Examine Region" :funcall selection-region)
                        ("Plan Path" :funcall selection-plan)
                        ("Reset" :funcall selection-reset)
                        ("Quit" :funcall selection-quit)))) t)


(defun create-map-menu-window ()
  (setf *map-menu-window*
        (tv: make-window
          'tv:pop-up-menu
            ':label "Map Selection"
            ':borders 3
            ':item-list '(("Synthetic Terrain-1"
                           :eval (selection-map-level-1
                                   "MAP-SYNTHETIC-TERRAIN-1"))
```

```
                          ("Synthetic Terrain-2"
                           :eval (selection-map-level-1
                                   "MAP-SYNTHETIC-TERRAIN-2")
                          ("Synthetic Terrain-3"
                           :eval (selection-map-level-1
                                   "MAP-SYNTHETIC-TERRAIN-3"))
                          ("Fort Hunter Liggett-HR"
                           :eval (selection-map-level-1
                                   "MAP-FORT-HUNTER-LIGGETT-HR"))
                          ("Fort Hunter Liggett-LR"
                           :eval (selection-map-level-1
                                   "MAP-FORT-HUNTER-LIGGETT-HR"))))) t)


(defun create-vehicle-menu-window ()
  (setf *vehicle-menu-window*
        (tv: make-window
          'tv:pop-up-menu
            ':label "Vehicle Selection"
            ':borders 3
            ':item-list '(("M113 Armored Personnel Carrier"
                           :eval (selection-vehicle-level-1 'm113-apc))
                          ("M966 Armored Tow Carrier"
                           :eval (selection-vehicle-level-1 'm966-atc))
                          ("M813 Cargo Truck"
                           :eval (selection-vehicle-level-1 'm813-ct))))) t)


(defun create-mission-menu-window ()
  (setf *mission-menu-window*
        (tv: make-window
          'tv:pop-up-menu
            ':label "Mission Selection"
            ':borders 3
            ':item-list '(("Assert Start Location"
                           :funcall selection-mission-start)
                          ("Assert Goal Location"
                           :funcall selection-mission-goal)))) t)


(defun initialize-terrain-window ()
  (send *terrain-window* :draw-fat-line 30 30 589 30)
  (send *terrain-window* :draw-fat-line 30 30 30 589)
  (send *terrain-window* :draw-fat-line 30 589 589 589)
  (send *terrain-window* :draw-fat-line 589 30 589 589)
  (send *terrain-window* :draw-fat-line 30 600 589 600)
  (send *terrain-window* :draw-fat-line 30 600 30 700)
  (send *terrain-window* :draw-fat-line 589 600 589 700)
  (send *terrain-window* :draw-fat-line 30 700 589 700)
  (send *terrain-window* :draw-string "TERRAIN MAP SUMMARY" 225 620)
  (send *terrain-window* :draw-string "ISOTROPIC REGIONS: " 40 640)
  (send *terrain-window* :draw-string "ANISOTROPIC-SAFE REGIONS: " 40 655)
  (send *terrain-window* :draw-string
                          "ANISOTROPIC-PARTIALLY-SAFE REGIONS: " 40 670)
  (send *terrain-window* :draw-string "OBSTACLE REGIONS: " 40 685)
  (send *terrain-window* :draw-string "VERTEX WINDOWS: " 375 640)
  (send *terrain-window* :draw-string "EDGE WINDOWS: " 375 655) t)


(defun initialize-profile-window ()
```

```
(send *profile-window* :draw-fat-line 20 20 375 20)
(send *profile-window* :draw-fat-line 20 20 20 185)
(send *profile-window* :draw-fat-line 375 20 375 185)
(send *profile-window* :draw-fat-line 20 185 375 185)
(send *profile-window* :draw-string  "VEHICLE PROFILE" 140 40)
(send *profile-window* :draw-string "DESIGNATION:" 40 60)
(send *profile-window* :draw-string "TYPE:" 225 60)
(send *profile-window* :draw-string "WEIGHT:" 40 90)
(send *profile-window* :draw-string "(LBS)" 325 90)
(send *profile-window* :draw-string "GRADIENT SLOPE:" 40 105)
(send *profile-window* :draw-string "(DEG)" 325 105)
(send *profile-window* :draw-string "CONTOUR SLOPE:" 40 120)
(send *profile-window* :draw-string "(DEG)" 325 120)
(send *profile-window* :draw-string "COASTING SLOPE:" 40 135)
(send *profile-window* :draw-string "(DEG)" 325 135)
(send *profile-window* :draw-string "MOTION RESISTANCE COEFF (min):" 40 155)
(send *profile-window* :draw-string "MOTION RESISTANCE COEFF (max):" 40 170)
(send *profile-window* :draw-fat-line 20 200 375 200)
(send *profile-window* :draw-fat-line 20 200 20 460)
(send *profile-window* :draw-fat-line 375 200 375 460)
(send *profile-window* :draw-fat-line 20 460 375 460)
(send *profile-window* :draw-string  "REGION PROFILE" 140 220)
(send *profile-window* :draw-string "DESIGNATION:" 40 240)
(send *profile-window* :draw-string "TYPE:" 210 240)
(send *profile-window* :draw-string "GRADIENT SLOPE:" 40 270)
(send *profile-window* :draw-string "(DEG)" 325 270)
(send *profile-window* :draw-string "ORIENTATION:" 40 285)
(send *profile-window* :draw-string "(DEG)" 325 285)
(send *profile-window* :draw-string "STABILITY CONSTRAINTS:" 40 300)
(send *profile-window* :draw-string "Gradient (up):" 52 315)
(send *profile-window* :draw-string "(DEG)" 325 315)
(send *profile-window* :draw-string "Gradient (dn):" 52 330)
(send *profile-window* :draw-string "(DEG)" 325 330)
(send *profile-window* :draw-string "BRAKING CONSTRAINTS:" 40 345)
(send *profile-window* :draw-string "Gradient (dn):" 52 360)
(send *profile-window* :draw-string "(DEG)" 325 360)
(send *profile-window* :draw-string "SURFACE COMPOSITION:" 40 375)
(send *profile-window* :draw-string "SURFACE COVERING:" 40 390)
(send *profile-window* :draw-string "SUB-TYPE:" 40 405)
(send *profile-window* :draw-string "SIDE VIEW:" 40 430)
(send *profile-window* :draw-fat-line 20 475 375 475)
(send *profile-window* :draw-fat-line 20 475 20 540)
(send *profile-window* :draw-fat-line 375 475 375 540)
(send *profile-window* :draw-fat-line 20 540 375 540)
(send *profile-window* :draw-string  "MISSION PROFILE" 140 495)
(send *profile-window* :draw-string "START  X:      Y:     ELEVATION:" 40 515)
(send *profile-window* :draw-string "GOAL   X:      Y:     ELEVATION:" 40 530)
(send *profile-window* :draw-fat-line 20 555 375 555)
(send *profile-window* :draw-fat-line 20 555 20 700)
(send *profile-window* :draw-fat-line 375 555 375 700)
(send *profile-window* :draw-fat-line 20 700 375 700)
(send *profile-window* :draw-string  "PLANNING PROFILE" 140 575)
(send *profile-window* :draw-string "CURRENT AGENDA LENGTH:" 40 595)
(send *profile-window* :draw-string "LOCAL OPTIMAL PATH COST:" 40 610)
(send *profile-window* :draw-string "TOTAL LOCAL OPTIMAL PATHS:" 40 625)
(send *profile-window* :draw-string "GLOBAL OPTIMAL PATH COST:" 40 655)
(send *profile-window* :draw-string "GLOBAL PLANNING TIME:" 40 670)
(send *profile-window* :draw-string "(MIN)" 325 670) t)
```

```
(defun set-display ()
  (create-terrain-window)
  (initialize-terrain-window)
  (create-profile-window)
  (initialize-profile-window)
  (create-top-level-menu-window)
  (create-map-menu-window)
  (create-vehicle-menu-window)
  (create-mission-menu-window) t)


(defun reset-display ()
  (send *terrain-window* :clear-window)
  (send *profile-window* :clear-window)
  (initialize-terrain-window)
  (initialize-profile-window)
  (display-map)
  (display-map-summary) t)


(defun refresh-display ()
  (send *terrain-window* :clear-window)
  (send *profile-window* :clear-window)
  (initialize-terrain-window)
  (initialize-profile-window)
  (display-map)
  (display-map-summary)
  (display-vehicle)
  (display-mission)
  (display-region)
  (display-statistics) t)


(defun refresh-display-with-local-path ()
  (send *terrain-window* :clear-window)
  (send *profile-window* :clear-window)
  (initialize-terrain-window)
  (initialize-profile-window)
  (display-map)
  (display-map-summary)
  (display-vehicle)
  (display-mission)
  (display-region)
  (display-statistics)
  (display-local-path) t)


(defun refresh-display-with-global-path ()
  (send *terrain-window* :clear-window)
  (send *profile-window* :clear-window)
  (initialize-terrain-window)
  (initialize-profile-window)
  (display-map)
  (display-map-summary)
  (display-vehicle)
  (display-mission)
  (display-region)
  (display-statistics)
  (display-global-path) t)
```

```
(defun kill-display ()
  (send *terrain-window* :kill)
  (send *profile-window* :kill) t)


(defun delay-loop ()
  (do ((value 0 (1+ value)))
      ((= value 500000))) t)


(defun selection-map ()
  (send *map-menu-window* ':expose-near '(:mouse))
  (send *map-menu-window* ':choose) t)


(defun selection-vehicle ()
  (send *vehicle-menu-window* ':expose-near '(:mouse))
  (send *vehicle-menu-window* ':choose) t)


(defun selection-mission ()
  (send *mission-menu-window* ':expose-near '(:mouse))
  (send *mission-menu-window* ':choose) t)


(defun selection-region ()
  (send *top-level-menu-window* ':deactivate)
  (assert-region)
  (refresh-display) t)


(defun selection-plan ()
  (send *top-level-menu-window* ':deactivate)
  (plan-path) t)


(defun selection-reset ()
  (send *top-level-menu-window* ':deactivate)
  (mpp-initialize-with-current-map)
  (reset-display) t)


(defun selection-quit ()
  (setf *termination-flag* t))


(defun selection-map-level-1 (filename)
  (send *map-menu-window* ':deactivate)
  (send *top-level-menu-window* ':deactivate)
  (assert-map filename)
  (refresh-display) t)


(defun selection-vehicle-level-1 (vehicle)
  (send *vehicle-menu-window* ':deactivate)
  (send *top-level-menu-window* ':deactivate)
  (assert-vehicle vehicle)
  (assert-region-totals)
  (refresh-display) t)
```

```
(defun selection-mission-start ()
  (send *mission-menu-window* ':deactivate)
  (send *top-level-menu-window* ':deactivate)
  (assert-start)
  (refresh-display) t)


(defun selection-mission-goal ()
  (send *mission-menu-window* ':deactivate)
  (send *top-level-menu-window* ':deactivate)
  (assert-goal)
  (refresh-display) t)


(defun mpp-top-level-menu-window ()
  (send *top-level-menu-window* ':expose-near '(:mouse))
  (send *top-level-menu-window* ':choose)
  (send *top-level-menu-window* ':deactivate))


(defun display-map ()
  (if *current-map*
      (dolist (edge *edge-list*)
        (draw-edge edge))) t)


(defun display-map-summary ()
  (send *terrain-window* :draw-string
        (write-to-string *total-isotropic-regions*) 187 640)
  (send *terrain-window* :draw-string
        (write-to-string *total-anisotropic-safe-regions*) 243 655)
  (send *terrain-window* :draw-string
        (write-to-string *total-anisotropic-partially-safe-regions*) 323 670)
  (send *terrain-window* :draw-string
        (write-to-string *total-obstacle-regions*) 180 685)
  (send *terrain-window* :draw-string
        (write-to-string (length *vertex-list*)) 498 640)
  (send *terrain-window* :dra-string
        (write-to-string (+    ngth *edge-list*)
                         (le.   n *background-edge-list*))) 482 655) t)


(defun display-vehicle ()
  (if *current-vehicle*
     (let ()
       (send *profile-window* :draw-string
             (write-to-string *current-vehicle*) 140 60)
       (send *profile-window* :draw-string
             (write-to-string (vehicle-type (eval *current-vehicle*))) 268 60)
       (send *profile-window* :draw-string
             (write-to-string (vehicle-weight (eval *current-vehicle*))) 101 90)
       (send *profile-window* :draw-string
             (write-to-string (vehicle-gradient-slope
                                (eval *current-vehicle*))) 163 105)
       (send *profile-window* :draw-string
             (write-to-string (vehicle-contour-slope
                                (eval *current-vehicle*))) 155 120)
       (send *profile-window* :draw-string
             (write-to-string (vehicle-coasting-slope
                                (eval *current-vehicle*))) 163 135)
```

```
        (send *profile-window* :draw-string
              (write-to-string *motion-resistance-lower*) 283 155)
        (send *profile-window* :draw-string
              (write-to-string *motion-resistance-upper*) 283 170))) t)


(defun display-mission ()
  (let ()
    (if *start-location*
        (let ((xcoord (truncate (first *start-location*)))
              (ycoord (truncate (second *start-location*)))
              (zcoord (third *start-location*)))
          (send *terrain-window*
                :draw-filled-in-circle (+ 30 xcoord) (- 589 ycoord) 2)
          (send *terrain-window*
                :draw-string "S" (+ 35 xcoord)  (- 589 ycoord))
          (send *profile-window*
                :draw-string (write-to-string xcoord) 116 515)
          (send *profile-window*
                :draw-string (write-to-string ycoord) 180 515)
          (send *profile-window*
                :draw-string (write-to-string zcoord) 300 515)))
    (if *goal-location*
        (let ((xcoord (truncate (first *goal-location*)))
              (ycoord (truncate (second *goal-location*)))
              (zcoord (third *goal-location*)))
          (send *terrain-window*
                :draw-filled-in-circle (+ 30 xcoord) (- 589 ycoord) 2)
          (send *terrain-window*
                :draw-string "G" (+ 35 xcoord) (- 589 ycoord))
          (send *profile-window*
                :draw-string (write-to-string xcoord) 116 530)
          (send *profile-window*
                :draw-string (write-to-string ycoord) 180 530)
          (send *profile-window*
                :draw-string (write-to-string zcoord) 300 530))))
    (delay-loop) t)


(defun display-region ()
  (if (and *current-region-examined* *current-vehicle*)
      (if (obstacle-region-p *current-region-examined*)
          (let ()
            (send *profile-window* :draw-string
                  (write-to-string *current-region-examined*) 140 240)
            (send *profile-window* :draw-string "OBSTACLE" 253 240)
            (send *profile-window* :draw-string "> CBR" 163 270)
            (send *profile-window* :draw-string "NIL" 139 285)
            (send *profile-window* :draw-string "NIL" 167 315)
            (send *profile-window* :draw-string "NIL" 167 330)
            (send *profile-window* :draw-string "NIL" 167 360)
            (send *profile-window* :draw-string "NIL" 204 375)
            (send *profile-window* :draw-string "NIL" 182 390)
            (send *profile-window* :draw-string "NIL" 116 405)
            (send *profile-window* :draw-rectangle 75 35 145 415) t)
          (let* ((rslope (make-significant-figures
                  (region-slope (eval *current-region-examined*)) 2))
                 (stability-constraints
                   (if (anisotropic-partially-safe-region-p
                         *current-region-examined*)
```

```
            (region-stability-constraints
              (eval *current-region-examined*))))
      (stability-constraints-revised
        (if stability-constraints
            (mapcar '(lambda (constraint)
              (make-significant-figures constraint 2))
                stability-constraints)))
      (braking-constraints
        (if (or (anisotropic-safe-region-p
                  *current-region-examined*)
                (anisotropic-partially-safe-region-p
                  *current-region-examined*))
      (region-braking-constraints (eval *current-region-examined*))))
      (braking-constraints-revised
        (if braking-constraints
            (mapcar '(lambda (constraint)
              (make-significant-figures constraint 2))
                braking-constraints)))
      (stability-gradient-down
        (if stability-constraints-revised
            (list (first stability-constraints-revised)
                  (second stability-constraints-revised))))
      (stability-gradient-up
        (if stability-constraints-revised
            (list (third stability-constraints-revised)
                  (fourth stability-constraints-revised))))
      (region-type nil)
      (region-subtype nil))
(if (and (isotropic-region-p *current-region-examined*)
         (null (member *current-region-examined*
                       *background-region-list*)))
    (let ()
      (setf region-type 'isotropic)
      (setf region-subtype 'nested))
    (if (equal (region-type
                 (eval *current-region-examined*)) 'isotropic)
        (let ()
          (setf region-type 'isotropic)
          (setf region-subtype 'background))
        (if (equal (region-type (eval *current-region-examined*))
                   'anisotropic-safe)
            (let ()
              (setf region-type 'anisotropic)
              (setf region-subtype 'safe))
            (if (equal (region-type
                         (eval *current-region-examined*))
                       'anisotropic-partially-safe)
                (let ()
                  (setf region-type 'anisotropic)
                  (setf region-subtype 'partially-safe))))))
(send *profile-window* :draw-string
      (write-to-string *current-region-examined*) 140 240)
(send *profile-window* :draw-string
      (write-to-string region-type) 253 240)
(send *profile-window* :draw-string
      (write-to-string rslope) 164 270)
(send *profile-window* :draw-string
      (write-to-string (region-orientation
                         (eval *current-region-examined*))) 139 285)
(send *profile-window* :draw-string
```

```
                (write-to-string stability-gradient-up) 167 315)
          (send *profile-window* :draw-string
                (write-to-string stability-gradient-down) 167 330)
          (send *profile-window* :draw-string
                (write-to-string braking-constraints-revised) 167 360)
          (send *profile-window* :draw-string
                (write-to-string (region-surface-material
                                  (eval *current-region-examined*))) 204 375)
          (send *profile-window* :draw-string
                (write-to-string (region-surface-covering
                                  (eval *current-region-examined*))) 182 390)
          (send *profile-window* :draw-string
                (write-to-string region-subtype) 116 405)
          (if (= rslope 0.0)
              (send *profile-window*
                    :draw-fat-line 130 450 200 450)
              (send *profile-window*
                    :draw-triangle 130 450 200 450 200
                      (- 450 (truncate (* (tan (degrees-to-radians rslope))
                        70)))))))) t)


(defun display-statistics ()
  (if *agenda-length*
      (send *profile-window* :draw-string
            (write-to-string *agenda-length*) 220 595))
  (if *local-optimal-path-cost*
      (send *profile-window* :draw-string
            (write-to-string
              (make-significant-figures *local-optimal-path-cost* 2)) 236 610))
  (if (> *total-local-optimal-paths* 0)
      (send *profile-window* :draw-string
            (write-to-string *total-local-optimal-paths*) 252 625))
  (if *global-optimal-path-cost*
      (send *profile-window* :draw-string
            (write-to-string
              (make-significant-figures *global-optimal-path-cost* 2)) 244 655))
  (if *global-planning-time*
      (send *profile-window* :draw-string
            (write-to-string
              (make-significant-figures
                (/ *global-planning-time* 60.0) 2)) 212 670)) t)


(defun display-local-path ()
  (if *local-optimal-path*
      (draw-path *local-optimal-path*)) t)


(defun display-global-path ()
  (if *global-optimal-path*
      (draw-path *global-optimal-path*)) t)


(defun draw-edge (edge)
  (let* ((s-edge (eval edge))
         (vertex-list (edge-vertex-list s-edge))
         (vertex1 (eval (first vertex-list)))
         (vertex2 (eval (second vertex-list))))
    (send *terrain-window* :draw-line
```

```
          (+ 30 (truncate (vertex-x-coord vertex1)))
          (- 589 (truncate (vertex-y-coord vertex1)))
          (+ 30 (truncate (vertex-x-coord vertex2)))
          (- 589 (truncate (vertex-y-coord vertex2))))) t)


(defun draw-thick-edge (edge)
  (let* ((s-edge (eval edge))
         (vertex-list (edge-vertex-list s-edge))
         (vertex1 (eval (first vertex-list)))
         (vertex2 (eval (second vertex-list))))
    (send *terrain-window* :draw-fat-line
          (+ 30 (truncate (vertex-x-coord vertex1)))
          (- 589 (truncate (vertex-y-coord vertex1)))
          (+ 30 (truncate (vertex-x-coord vertex2)))
          (- 589 (truncate (vertex-y-coord vertex2))))) t)


(defun draw-path-segment (coord-list1 coord-list2)
  (let ((x1 (first coord-list1))
        (y1 (second coord-list1))
        (x2 (first coord-list2))
        (y2 (second coord-list2)))
    (send *terrain-window* :draw-fat-line
          (+ 30 (truncate x1))
          (- 589 (truncate y1))
          (+ 30 (truncate x2))
          (- 589 (truncate y2)))) t)


(defun draw-path (path-coord-list)
  (do ((coord-list path-coord-list (rest coord-list)))
      ((= (length coord-list) 1) t)
    (let* ((coord-list1 (first  d-list))
           (coord-list2 (second coord-list)))
      (draw-path-segment coord-list1 coord-list2))) t)


(defun get-mouse-coordinates ()
  (let* ((mouse-blip (send *terrain-window* :list-tyi))
         (xcoord (- (fourth mouse-blip) 40))
         (ycoord (- 599 (fifth mouse-blip))))
    (list xcoord ycoord)))


(defun special-effects ()
  (let ((original-brightness-level (tv:screen-brightness tv:main-screen)))
    (dotimes (counter 20)
      (dotimes (counter 10000))
      (setf (tv:screen-brightness tv:main-screen) 0)
      (dotimes (counter 10000))
      (setf (tv:screen-brightness tv:main-screen) 1))
    (setf (tv:screen-brightness tv:main-screen) original-brightness-level)) t)

;*******************************************************************************


(defvar *terrain-window*)
(defvar *profile-window*)
```

```
(defvar *top-level-menu-window*)
(defvar *map-menu-window*)
(defvar *vehicle-menu-window*)
(defvar *mission-menu-window*)


(setf *terrain-window* nil)
(setf *profile-window* nil)
(setf *top-level-menu-window* nil)
(setf *map-menu-window* nil)
(setf *vehicle-menu-window* nil)
(setf *mission-menu-window* nil)


;*******************************************************************************
```

APPENDIX B - SYNTHETIC TERRAIN STRUCTURES

```
;*****************************************************************************
;
; Global Vertex List:
;
;*****************************************************************************

(V-58 V-57 V-56 V-55 V-54 V-53 V-52 V-51 V-50 V-49 V-48 V-47 V-24 V-23 V-22 V-21
 V-20 V-19 V-18 V-17 V-16 V-15 V-14 V-13 V-12 V-11 V-10 V-9 V-8 V-7 V-6 V-5 V-4
 V-3 V-2 V-1 V-NW V-NE V-SE V-SW)

;*****************************************************************************
;
; Global Edge List:
;
;*****************************************************************************

(E-86 E-85 E-84 E-83 E-82 E-81 E-80 E-79 E-78 E-77 E-76 E-75 E-74 E-73 E-72 E-71
 E-70 E-69 E-68 E-67 E-66 E-65 E-64 E-63 E-40 E-39 E-38 E-37 E-36 E-35 E-34 E-33
 E-32 E-31 E-30 E-29 E-28 E-27 E-26 E-25 E-24 E-23 E-22 E-21 E-20 E-19 E-18 E-17
 E-16 E-15 E-14 E-13 E-12 E-11 E-10 E-9 E-8 E-7 E-6 E-5 E-4 E-3 E-2 E-1 E-N E-S
 E-E E-W)

;*****************************************************************************
;
; Global Background Edge List:
;
;*****************************************************************************

(BE-11 BE-10 BE-9 BE-8 BE-7 BE-6 BE-5 BE-4 BE-3 BE-2 BE-1)

;*****************************************************************************
;
; Global Region List:
;
;*****************************************************************************

(R-35 R-34 R-33 R-32 R-31 R-30 R-29 R-28 R-27 R-26 R-25 R-24 R-18 R-17 R-16 R-15
 R-14 R-13 R-12 R-11 R-10 R-9 R-8 R-7 R-6 R-5 R-4 R-3 R-2 R-1)

;*****************************************************************************
;
; Global Background Region List:
;
;*****************************************************************************

(BR-10 BR-9 BR-8 BR-7 BR-6 BR-5 BR-4 BR-3 BR-2 BR-1)

;*****************************************************************************
```

```
;*****************************************************************************
;
; Vertex Structures:
;
;*****************************************************************************


#S(VERTEX :X-COORD 352.0
          :Y-COORD 96.0
          :Z-COORD 14.5
          :EDGE-LIST (E-36 E-79 E-82 E-86)
          :VISIBILITY-LIST (V-13 V-15 V-16 V-17 V-55 V-19 V-57 V-20))

#S(VERTEX :X-COORD 480.0
          :Y-COORD 224.0
          :Z-COORD 14.5
          :EDGE-LIST (E-35 E-81 E-82 E-85)
          :VISIBILITY-LIST (V-14 V-15 V-16 V-18 V-56 V-19 V-20 V-58))

#S(VERTEX :X-COORD 400.0
          :Y-COORD 304.0
          :Z-COORD 14.5
          :EDGE-LIST (E-34 E-80 E-81 E-84)
          :VISIBILITY-LIST (V-13 V-14 V-15 V-17 V-55 V-18 V-19 V-57))

#S(VERTEX :X-COORD 272.0
          :Y-COORD 176.0
          :Z-COORD 14.5
          :EDGE-LIST (E-33 E-79 E-80 E-83)
          :VISIBILITY-LIST (V-16 V-13 V-14 V-20 V-58 V-17 V-18 V-56))

#S(VERTEX :X-COORD 240.0
          :Y-COORD 320.0
          :Z-COORD 9.0
          :EDGE-LIST (E-20 E-73 E-74 E-78)
          :VISIBILITY-LIST (V-11 V-12 V-9 V-5 V-51 V-8 V-7 V-53))

#S(VERTEX :X-COORD 240.0
          :Y-COORD 448.0
          :Z-COORD 9.0
          :EDGE-LIST (E-19 E-72 E-73 E-77)
          :VISIBILITY-LIST (V-10 V-11 V-12 V-6 V-52 V-8 V-54 V-7))

#S(VERTEX :X-COORD 112.0
          :Y-COORD 448.0
          :Z-COORD 9.0
          :EDGE-LIST (E-18 E-71 E-72 E-76)
          :VISIBILITY-LIST (V-9 V-10 V-11 V-5 V-51 V-7 V-53 V-6))

#S(VERTEX :X-COORD 112.0
          :Y-COORD 320.0
          :Z-COORD 9.0
          :EDGE-LIST (E-17 E-71 E-74 E-75)
          :VISIBILITY-LIST (V-10 V-12 V-9 V-8 V-54 V-6 V-52 V-5))

#S(VERTEX :X-COORD 192.0
          :Y-COORD 368.0
          :Z-COORD 27.75
          :EDGE-LIST (E-63 E-66 E-70)
          :VISIBILITY-LIST (V-47 V-9 V-49 V-12 V-11 V-48))
```

```
#S(VERTEX :X-COORD 160.0
          :Y-COORD 368.0
          :Z-COORD 27.75
          :EDGE-LIST (E-63 E-64 E-67)
          :VISIBILITY-LIST (V-48 V-12 V-50 V-10 V-47 V-9))
```

```
#S(VERTEX :X-COORD 192.0
          :Y-COORD 400.0
          :Z-COORD 27.75
          :EDGE-LIST (E-65 E-66 E-69)
          :VISIBILITY-LIST (V-49 V-10 V-47 V-12 V-50 V-11))

#S(VERTEX :X-COORD 160.0
          :Y-COORD 400.0
          :Z-COORD 27.75
          :EDGE-LIST (E-64 E-65 E-68)
          :VISIBILITY-LIST (V-50 V-9 V-49 V-11 V-48 V-10))

#S(VERTEX :X-COORD 352.0
          :Y-COORD 160.0
          :Z-COORD 35.75
          :EDGE-LIST (E-32 E-29 E-40)
          :VISIBILITY-LIST (V-22 V-17 V-21 V-20 V-19 V-23))

#S(VERTEX :X-COORD 416.0
          :Y-COORD 224.0
          :Z-COORD 35.75
          :EDGE-LIST (E-31 E-32 E-39)
          :VISIBILITY-LIST (V-21 V-18 V-22 V-20 V-24 V-19))

#S(VERTEX :X-COORD 400.0
          :Y-COORD 240.0
          :Z-COORD 35.75
          :EDGE-LIST (E-30 E-31 E-38)
          :VISIBILITY-LIST (V-24 V-17 V-21 V-19 V-23 V-18))

#S(VERTEX :X-COORD 336.0
          :Y-COORD 176.0
          :Z-COORD 35.75
          :EDGE-LIST (E-29 E-30 E-37)
          :VISIBILITY-LIST (V-23 V-20 V-24 V-18 V-22 V-17))

#S(VERTEX :X-COORD 352.0
          :Y-COORD 112.0
          :Z-COORD 18.0
          :EDGE-LIST (E-25 E-28 E-40 E-86)
          :VISIBILITY-LIST (V-21 V-24 V-23 V-17 V-55 V-19 V-57 V-58))

#S(VERTEX :X-COORD 464.0
          :Y-COORD 224.0
          :Z-COORD 18.0
          :EDGE-LIST (E-27 E-28 E-39 E-85)
          :VISIBILITY-LIST (V-22 V-24 V-23 V-18 V-56 V-57 V-20 V-58))

#S(VERTEX :X-COORD 400.0
          :Y-COORD 288.0
          :Z-COORD 18.0
          :EDGE-LIST (E-26 E-27 E-38 E-84)
          :VISIBILITY-LIST (V-21 V-23 V-22 V-17 V-55 V-56 V-19 V-57))

#S(VERTEX :X-COORD 288.0
          :Y-COORD 176.0
          :Z-COORD 18.0
          :EDGE-LIST (E-25 E-26 E-37 E-83)
          :VISIBILITY-LIST (V-24 V-22 V-21 V-20 V-58 V-55 V-18 V-56))
```

```
#S(VERTEX :X-COORD 352.0
          :Y-COORD 48.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-9 BE-10 E-21 E-24 E-36)
          :VISIBILITY-LIST (V-SW V-SE V-13 V-55 V-15 V-57 V-58))
```

```
#S(VERTEX :X-COORD 528.0
          :Y-COORD 224.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-5 BE-11 E-23 E-24 E-35)
          :VISIBILITY-LIST (V-NE V-SE V-14 V-56 V-16 V-57 V-58))

#S(VERTEX :X-COORD 400.0
          :Y-COORD 352.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-4 BE-6 E-22 E-23 E-34)
          :VISIBILITY-LIST (V-3 V-4 V-NE V-13 V-55 V-15 V-56 V-57))

#S(VERTEX :X-COORD 224.0
          :Y-COORD 176.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-7 BE-8 E-21 E-22 E-33)
          :VISIBILITY-LIST (V-1 V-SW V-4 V-16 V-58 V-14 V-55 V-56))

#S(VERTEX :X-COORD 208.0
          :Y-COORD 352.0
          :Z-COORD 19.5
          :EDGE-LIST (E-9 E-12 E-20 E-70)
          :VISIBILITY-LIST (V-51 V-53 V-54 V-9 V-49 V-50 V-11 V-48))

#S(VERTEX :X-COORD 208.0
          :Y-COORD 416.0
          :Z-COORD 19.5
          :EDGE-LIST (E-11 E-12 E-19 E-69)
          :VISIBILITY-LIST (V-52 V-53 V-54 V-10 V-47 V-12 V-50 V-48))

#S(VERTEX :X-COORD 144.0
          :Y-COORD 416.0
          :Z-COORD 19.5
          :EDGE-LIST (E-10 E-11 E-18 E-68)
          :VISIBILITY-LIST (V-51 V-52 V-53 V-9 V-49 V-11 V-48 V-47))

#S(VERTEX :X-COORD 144.0
          :Y-COORD 352.0
          :Z-COORD 19.5
          :EDGE-LIST (E-9 E-10 E-17 E-67)
          :VISIBILITY-LIST (V-54 V-51 V-52 V-12 V-50 V-10 V-47 V-49))

#S(VERTEX :X-COORD 256.0
          :Y-COORD 304.0
          :Z-COORD 9.0
          :EDGE-LIST (E-5 E-8 E-16 E-78)
          :VISIBILITY-LIST (V-1 V-4 V-3 V-5 V-51 V-54 V-7 V-53))

#S(VERTEX :X-COORD 256.0
          :Y-COORD 464.0
          :Z-COORD 9.0
          :EDGE-LIST (E-7 E-8 E-15 E-77)
          :VISIBILITY-LIST (V-2 V-4 V-3 V-6 V-52 V-8 V-54 V-53))

#S(VERTEX :X-COORD 96.0
          :Y-COORD 464.0
          :Z-COORD 9.0
          :EDGE-LIST (E-6 E-7 E-14 E-76)
          :VISIBILITY-LIST (V-1 V-3 V-2 V-5 V-51 V-7 V-53 V-52))
```

```
#S(VERTEX :X-COORD 96.0
          :Y-COORD 304.0
          :Z-COORD 9.0
          :EDGE-LIST (E-5 E-6 E-13 E-75)
          :VISIBILITY-LIST (V-4 V-2 V-1 V-8 V-54 V-6 V-52 V-51))
```

```
#S(VERTEX :X-COORD 288.0
          :Y-COORD 272.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-6 BE-7 E-1 E-4 E-16)
          :VISIBILITY-LIST (V-SW V-13 V-NE V-14 V-1 V-5 V-8 V-3 V-7))

#S(VERTEX :X-COORD 288.0
          :Y-COORD 496.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-3 E-3 E-4 E-15)
          :VISIBILITY-LIST (V-NW V-NE V-14 V-2 V-6 V-4 V-8 V-7))

#S(VERTEX :X-COORD 64.0
          :Y-COORD 496.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-2 E-2 E-3 E-14)
          :VISIBILITY-LIST (V-SW V-NW V-NE V-1 V-5 V-3 V-7 V-6))

#S(VERTEX :X-COORD 64.0
          :Y-COORD 272.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-1 E-1 E-2 E-13)
          :VISIBILITY-LIST (V-13 V-SW V-NW V-4 V-8 V-2 V-6 V-5))

#S(VERTEX :X-COORD 0.0
          :Y-COORD 559.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-2 E-W E-N)
          :VISIBILITY-LIST (V-SW V-1 V-NE V-2 V-3))

#S(VERTEX :X-COORD 559.0
          :Y-COORD 559.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-3 BE-4 BE-5 E-N E-E)
          :VISIBILITY-LIST (V-4 V-14 V-NW V-2 V-3 V-SE V-15))

#S(VERTEX :X-COORD 559.0
          :Y-COORD 0.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-10 BE-11 E-S E-E)
          :VISIBILITY-LIST (V-16 V-SW V-NE V-15))

#S(VERTEX :X-COORD 0.0
          :Y-COORD 0.0
          :Z-COORD 0.0
          :EDGE-LIST (BE-1 BE-8 BE-9 E-W E-S)
          :VISIBILITY-LIST (V-4 V-13 V-NW V-1 V-2 V-16 V-SE))
```

```
;*****************************************************************************
;
; Edge Structures:
;
;*****************************************************************************


#S(EDGE :VERTEX-LIST (V-20 V-58)
        :ADJACENCY-LIST (R-32 R-35)
        :VISIBILITY-LIST (E-25 E-79 E-83 E-28 E-85 E-82))

#S(EDGE :VERTEX-LIST (V-19 V-57)
        :ADJACENCY-LIST (R-34 R-35)
        :VISIBILITY-LIST (E-27 E-84 E-81 E-28 E-82 E-86))

#S(EDGE :VERTEX-LIST (V-18 V-56)
        :ADJACENCY-LIST (R-33 R-34)
        :VISIBILITY-LIST (E-26 E-83 E-80 E-27 E-81 E-85))

#S(EDGE :VERTEX-LIST (V-17 V-55)
        :ADJACENCY-LIST (R-32 R-33)
        :VISIBILITY-LIST (E-25 E-86 E-79 E-26 E-80 E-84))

#S(EDGE :VERTEX-LIST (V-57 V-58)
        :ADJACENCY-LIST (R-13 R-35)
        :VISIBILITY-LIST (E-35 E-24 E-36 E-28 E-85 E-86))

#S(EDGE :VERTEX-LIST (V-56 V-57)
        :ADJACENCY-LIST (R-12 R-34)
        :VISIBILITY-LIST (E-34 E-23 E-35 E-27 E-84 E-85))

#S(EDGE :VERTEX-LIST (V-55 V-56)
        :ADJACENCY-LIST (R-11 R-33)
        :VISIBILITY-LIST (E-33 E-22 E-34 E-26 E-83 E-84))

#S(EDGE :VERTEX-LIST (V-55 V-58)
        :ADJACENCY-LIST (R-10 R-32)
        :VISIBILITY-LIST (E-21 E-33 E-36 E-25 E-86 E-83))

#S(EDGE :VERTEX-LIST (V-8 V-54)
        :ADJACENCY-LIST (R-28 R-31)
        :VISIBILITY-LIST (E-5 E-75 E-74 E-8 E-73 E-77))

#S(EDGE :VERTEX-LIST (V-7 V-53)
        :ADJACENCY-LIST (R-30 R-31)
        :VISIBILITY-LIST (E-7 E-72 E-76 E-8 E-78 E-73))

#S(EDGE :VERTEX-LIST (V-6 V-52)
        :ADJACENCY-LIST (R-29 R-30)
        :VISIBILITY-LIST (E-6 E-71 E-75 E-7 E-77 E-72))

#S(EDGE :VERTEX-LIST (V-5 V-51)
        :ADJACENCY-LIST (R-28 R-29)
        :VISIBILITY-LIST (E-5 E-74 E-78 E-6 E-76 E-71))

#S(EDGE :VERTEX-LIST (V-51 V-54)
        :ADJACENCY-LIST (R-5 R-28)
        :VISIBILITY-LIST (E-9 E-20 E-17 E-5 E-75 E-78))

#S(EDGE :VERTEX-LIST (V-53 V-54)
```

```
        :ADJACENCY-LIST (R-8 R-31)
        :VISIBILITY-LIST (E-12 E-19 E-20 E-8 E-78 E-77))

#S(EDGE :VERTEX-LIST (V-52 V-53)
        :ADJACENCY-LIST (R-7 R-30)
        :VISIBILITY-LIST (E-11 E-18 E-19 E-7 E-77 E-76))
```

```
#S(EDGE :VERTEX-LIST (V-51 V-52)
        :ADJACENCY-LIST (R-6 R-29)
        :VISIBILITY-LIST (E-10 E-17 E-18 E-6 E-76 E-75))

#S(EDGE :VERTEX-LIST (V-12 V-50)
        :ADJACENCY-LIST (R-24 R-27)
        :VISIBILITY-LIST (E-9 E-67 E-63 E-12 E-66 E-69))

#S(EDGE :VERTEX-LIST (V-11 V-48)
        :ADJACENCY-LIST (R-26 R-27)
        :VISIBILITY-LIST (E-11 E-65 E-68 E-12 E-70 E-66))

#S(EDGE :VERTEX-LIST (V-10 V-47)
        :ADJACENCY-LIST (R-25 R-26)
        :VISIBILITY-LIST (E-10 E-64 E-67 E-11 E-69 E-65))

#S(EDGE :VERTEX-LIST (V-9 V-49)
        :ADJACENCY-LIST (R-24 R-25)
        :VISIBILITY-LIST (E-9 E-63 E-70 E-10 E-68 E-64))

#S(EDGE :VERTEX-LIST (V-48 V-50)
        :ADJACENCY-LIST (R-9 R-27)
        :VISIBILITY-LIST (E-63 E-64 E-65 E-12 E-70 E-69))

#S(EDGE :VERTEX-LIST (V-47 V-48)
        :ADJACENCY-LIST (R-9 R-26)
        :VISIBILITY-LIST (E-63 E-64 E-66 E-11 E-69 E-68))

#S(EDGE :VERTEX-LIST (V-47 V-49)
        :ADJACENCY-LIST (R-9 R-25)
        :VISIBILITY-LIST (E-63 E-65 E-66 E-10 E-68 E-67))

#S(EDGE :VERTEX-LIST (V-49 V-50)
        :ADJACENCY-LIST (R-9 R-24)
        :VISIBILITY-LIST (E-64 E-65 E-66 E-9 E-67 E-70))

#S(EDGE :VERTEX-LIST (V-20 V-24)
        :ADJACENCY-LIST (R-14 R-17)
        :VISIBILITY-LIST (E-25 E-37 E-29 E-28 E-32 E-39))

#S(EDGE :VERTEX-LIST (V-19 V-23)
        :ADJACENCY-LIST (R-16 R-17)
        :VISIBILITY-LIST (E-27 E-31 E-38 E-28 E-40 E-32))

#S(EDGE :VERTEX-LIST (V-18 V-22)
        :ADJACENCY-LIST (R-15 R-16)
        :VISIBILITY-LIST (E-26 E-30 E-37 E-27 E-39 E-31))

#S(EDGE :VERTEX-LIST (V-17 V-21)
        :ADJACENCY-LIST (R-14 R-15)
        :VISIBILITY-LIST (E-25 E-29 E-40 E-26 E-38 E-30))

#S(EDGE :VERTEX-LIST (V-16 V-58)
        :ADJACENCY-LIST (R-10 R-13)
        :VISIBILITY-LIST (E-21 E-33 E-79 E-35 E-24 E-82))

#S(EDGE :VERTEX-LIST (V-15 V-57)
        :ADJACENCY-LIST (R-12 R-13)
        :VISIBILITY-LIST (E-34 E-23 E-81 E-24 E-36 E-82))
```

```
#S(EDGE :VERTEX-LIST (V-14 V-56)
        :ADJACENCY-LIST (R-11 R-12)
        :VISIBILITY-LIST (E-33 E-22 E-80 E-23 E-35 E-81))

#S(EDGE :VERTEX-LIST (V-13 V-55)
        :ADJACENCY-LIST (R-10 R-11)
        :VISIBILITY-LIST (E-21 E-79 E-36 E-22 E-34 E-80))

#S(EDGE :VERTEX-LIST (V-23 V-24)
        :ADJACENCY-LIST (R-17 R-18)
        :VISIBILITY-LIST (E-28 E-40 E-39 E-29 E-30 E-31))

#S(EDGE :VERTEX-LIST (V-22 V-23)
        :ADJACENCY-LIST (R-16 R-18)
        :VISIBILITY-LIST (E-27 E-39 E-38 E-29 E-30 E-32))

#S(EDGE :VERTEX-LIST (V-21 V-22)
        :ADJACENCY-LIST (R-15 R-18)
        :VISIBILITY-LIST (E-26 E-38 E-37 E-29 E-31 E-32))

#S(EDGE :VERTEX-LIST (V-21 V-24)
        :ADJACENCY-LIST (R-14 R-18)
        :VISIBILITY-LIST (E-25 E-37 E-40 E-30 E-31 E-32))

#S(EDGE :VERTEX-LIST (V-19 V-20)
        :ADJACENCY-LIST (R-17 R-35)
        :VISIBILITY-LIST (E-40 E-32 E-39 E-85 E-82 E-86))

#S(EDGE :VERTEX-LIST (V-18 V-19)
        :ADJACENCY-LIST (R-16 R-34)
        :VISIBILITY-LIST (E-39 E-31 E-38 E-84 E-81 E-85))

#S(EDGE :VERTEX-LIST (V-17 V-18)
        :ADJACENCY-LIST (R-15 R-33)
        :VISIBILITY-LIST (E-38 E-30 E-37 E-83 E-80 E-84))

#S(EDGE :VERTEX-LIST (V-17 V-20)
        :ADJACENCY-LIST (R-14 R-32)
        :VISIBILITY-LIST (E-37 E-29 E-40 E-86 E-79 E-83))

#S(EDGE :VERTEX-LIST (V-15 V-16)
        :ADJACENCY-LIST (R-13 BR-8)
        :VISIBILITY-LIST (E-35 E-36 E-82 BE-10 BE-11))

#S(EDGE :VERTEX-LIST (V-14 V-15)
        :ADJACENCY-LIST (R-12 BR-6)
        :VISIBILITY-LIST (E-34 E-35 E-81 BE-4 BE-5))

#S(EDGE :VERTEX-LIST (V-13 V-14)
        :ADJACENCY-LIST (R-11 BR-5)
        :VISIBILITY-LIST (E-33 E-34 E-80 BE-7 BE-6))

#S(EDGE :VERTEX-LIST (V-13 V-16)
        :ADJACENCY-LIST (R-10 BR-10)
        :VISIBILITY-LIST (E-33 E-79 E-36 BE-8 BE-9))

#S(EDGE :VERTEX-LIST (V-12 V-54)
        :ADJACENCY-LIST (R-5 R-8)
        :VISIBILITY-LIST (E-9 E-74 E-17 E-12 E-19 E-73))
```

```
#S(EDGE :VERTEX-LIST (V-11 V-53)
        :ADJACENCY-LIST (R-7 R-8)
        :VISIBILITY-LIST (E-11 E-18 E-72 E-12 E-73 E-20))

#S(EDGE :VERTEX-LIST (V-10 V-52)
        :ADJACENCY-LIST (R-6 R-7)
        :VISIBILITY-LIST (E-10 E-17 E-71 E-11 E-72 E-19))

#S(EDGE :VERTEX-LIST (V-9 V-51)
        :ADJACENCY-LIST (R-5 R-6)
        :VISIBILITY-LIST (E-9 E-20 E-74 E-10 E-71 E-18))

#S(EDGE :VERTEX-LIST (V-4 V-8)
        :ADJACENCY-LIST (R-1 R-4)
        :VISIBILITY-LIST (E-1 E-13 E-5 E-4 E-8 E-15))

#S(EDGE :VERTEX-LIST (V-3 V-7)
        :ADJACENCY-LIST (R-3 R-4)
        :VISIBILITY-LIST (E-3 E-7 E-14 E-4 E-16 E-8))

#S(EDGE :VERTEX-LIST (V-2 V-6)
        :ADJACENCY-LIST (R-2 R-3)
        :VISIBILITY-LIST (E-2 E-6 E-13 E-3 E-15 E-7))

#S(EDGE :VERTEX-LIST (V-1 V-5)
        :ADJACENCY-LIST (R-1 R-2)
        :VISIBILITY-LIST (E-1 E-5 E-16 E-2 E-14 E-6))

#S(EDGE :VERTEX-LIST (V-11 V-12)
        :ADJACENCY-LIST (R-8 R-27)
        :VISIBILITY-LIST (E-19 E-73 E-20 E-70 E-66 E-69))

#S(EDGE :VERTEX-LIST (V-10 V-11)
        :ADJACENCY-LIST (R-7 R-26)
        :VISIBILITY-LIST (E-18 E-72 E-19 E-69 E-65 E-68))

#S(EDGE :VERTEX-LIST (V-9 V-10)
        :ADJACENCY-LIST (R-6 R-25)
        :VISIBILITY-LIST (E-17 E-71 E-18 E-68 E-64 E-67))

#S(EDGE :VERTEX-LIST (V-9 V-12)
        :ADJACENCY-LIST (R-5 R-24)
        :VISIBILITY-LIST (E-20 E-74 E-17 E-67 E-63 E-70))

#S(EDGE :VERTEX-LIST (V-7 V-8)
        :ADJACENCY-LIST (R-4 R-31)
        :VISIBILITY-LIST (E-4 E-16 E-15 E-78 E-73 E-77))

#S(EDGE :VERTEX-LIST (V-6 V-7)
        :ADJACENCY-LIST (R-3 R-30)
        :VISIBILITY-LIST (E-3 E-15 E-14 E-77 E-72 E-76))

#S(EDGE :VERTEX-LIST (V-5 V-6)
        :ADJACENCY-LIST (R-2 R-29)
        :VISIBILITY-LIST (E-2 E-14 E-13 E-76 E-71 E-75))

#S(EDGE :VERTEX-LIST (V-5 V-8)
        :ADJACENCY-LIST (R-1 R-28)
        :VISIBILITY-LIST (E-1 E-13 E-16 E-75 E-74 E-78))
```

```
#S(EDGE :VERTEX-LIST (V-3 V-4)
        :ADJACENCY-LIST (R-4 BR-4)
        :VISIBILITY-LIST (E-16 E-8 E-15 BE-3 BE-4 BE-6))

#S(EDGE :VERTEX-LIST (V-2 V-3)
        :ADJACENCY-LIST (R-3 BR-3)
        :VISIBILITY-LIST (E-15 E-7 E-14 BE-2 E-N BE-3))

#S(EDGE :VERTEX-LIST (V-1 V-2)
        :ADJACENCY-LIST (R-2 BR-2)
        :VISIBILITY-LIST (E-14 E-6 E-13 BE-1 E-W BE-2))

#S(EDGE :VERTEX-LIST (V-1 V-4)
        :ADJACENCY-LIST (R-1 BR-1)
        :VISIBILITY-LIST (E-13 E-5 E-16 BE-8 BE-1 BE-7))

#S(EDGE :VERTEX-LIST (V-NW V-NE)
        :ADJACENCY-LIST (BR-3)
        :VISIBILITY-LIST (BE-2 BE-3 E-3))

#S(EDGE :VERTEX-LIST (V-SE V-SW)
        :ADJACENCY-LIST (BR-9)
        :VISIBILITY-LIST (BE-9 BE-10))

#S(EDGE :VERTEX-LIST (V-NE V-SE)
        :ADJACENCY-LIST (BR-7)
        :VISIBILITY-LIST (BE-5 BE-11))

#S(EDGE :VERTEX-LIST (V-SW V-NW)
        :ADJACENCY-LIST (BR-2)
        :VISIBILITY-LIST (BE-1 BE-2 E-2))

#S(EDGE :VERTEX-LIST (V-SE V-15)
        :ADJACENCY-LIST (BR-7 BR-8)
        :VISIBILITY-LIST (BE-5 E-E BE-10 E-24))

#S(EDGE :VERTEX-LIST (V-SE V-16)
        :ADJACENCY-LIST (BR-8 BR-9)
        :VISIBILITY-LIST (E-24 BE-11 BE-9 E-S))

#S(EDGE :VERTEX-LIST (V-SW V-16)
        :ADJACENCY-LIST (BR-9 BR-10)
        :VISIBILITY-LIST (BE-10 E-S BE-8 E-21))

#S(EDGE :VERTEX-LIST (V-SW V-13)
        :ADJACENCY-LIST (BR-1 BR-10)
        :VISIBILITY-LIST (BE-1 E-1 BE-7 E-21 BE-9))

#S(EDGE :VERTEX-LIST (V-4 V-13)
        :ADJACENCY-LIST (BR-1 BR-5)
        :VISIBILITY-LIST (BE-8 BE-1 E-1 BE-6 E-22))

#S(EDGE :VERTEX-LIST (V-4 V-14)
        :ADJACENCY-LIST (BR-4 BR-5)
        :VISIBILITY-LIST (BE-3 BE-4 E-4 BE-7 E-22))

#S(EDGE :VERTEX-LIST (V-NE V-15)
        :ADJACENCY-LIST (BR-6 BR-7)
        :VISIBILITY-LIST (BE-4 E-23 E-E BE-11))
```

```
#S(EDGE :VERTEX-LIST (V-NE V-14)
        :ADJACENCY-LIST (BR-4 BR-6)
        :VISIBILITY-LIST (BE-3 BE-6 E-4 BE-5 E-23))

#S(EDGE :VERTEX-LIST (V-NE V-3)
        :ADJACENCY-LIST (BR-3 BR-4)
        :VISIBILITY-LIST (BE-2 E-N E-3 BE-4 BE-6 E-4))

#S(EDGE :VERTEX-LIST (V-NW V-2)
        :ADJACENCY-LIST (BR-2 BR-3)
        :VISIBILITY-LIST (BE-1 E-W E-2 E-N BE-3 E-3))

#S(EDGE :VERTEX-LIST (V-SW V-1)
        :ADJACENCY-LIST (BR-1 BR-2)
        :VISIBILITY-LIST (BE-8 E-1 BE-7 E-W BE-2 E-2))
```

```
;*******************************************************************************
;
; Region Structures:
;
;*******************************************************************************


#S(REGION :EDGE-LIST (E-28 E-85 E-82 E-86)
          :SLOPE 12.339
          :ORIENTATION 135.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (98.714 171.286))

#S(REGION :EDGE-LIST (E-27 E-84 E-81 E-85)
          :SLOPE 12.339
          :ORIENTATION 45.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (8.714 81.286))

#S(REGION :EDGE-LIST (E-26 E-83 E-80 E-84)
          :SLOPE 12.339
          :ORIENTATION 315.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (278.714 351.286))

#S(REGION :EDGE-LIST (E-25 E-86 E-79 E-83)
          :SLOPE 12.339
          :ORIENTATION 225.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (188.714 261.286))

#S(REGION :EDGE-LIST (E-8 E-78 E-73 E-77)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (E-7 E-77 E-72 E-76)
          :SLOPE 0.0
          :ORIENTATION NIL
```

```
:SURFACE-MATERIAL GRAVEL-CLAY
:SURFACE-CONDITION DRY
:SURFACE-COVERING BRUSH
:TYPE NIL
:STABILITY-CONSTRAINTS NIL
:BRAKING-CONSTRAINTS NIL)
```

```
#S(REGION :EDGE-LIST (E-6 E-76 E-71 E-75)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (E-5 E-75 E-74 E-78)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (E-12 E-70 E-66 E-69)
          :SLOPE 27.277
          :ORIENTATION 90.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (50.0 130.0 230.0 310.0)
          :BRAKING-CONSTRAINTS (19.997 160.003))

#S(REGION :EDGE-LIST (E-11 E-69 E-65 E-68)
          :SLOPE 27.277
          :ORIENTATION 0.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (320.0 40.0 140.0 220.0)
          :BRAKING-CONSTRAINTS (289.997 70.003))

#S(REGION :EDGE-LIST (E-10 E-68 E-64 E-67)
          :SLOPE 27.277
          :ORIENTATION 270.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (230.0 310.0 50.0 130.0)
          :BRAKING-CONSTRAINTS (199.997 340.003))

#S(REGION :EDGE-LIST (E-9 E-67 E-63 E-70)
          :SLOPE 27.277
          :ORIENTATION 180.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (140.0 220.0 320.0 40.0)
          :BRAKING-CONSTRAINTS (109.997 250.003))
```

```
#S(REGION :EDGE-LIST (E-29 E-30 E-31 E-32)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (E-28 E-40 E-32 E-39)
          :SLOPE 29.017
          :ORIENTATION 135.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (105.0 165.0 285.0 345.0)
          :BRAKING-CONSTRAINTS (63.535 206.465))

#S(REGION :EDGE-LIST (E-27 E-39 E-31 E-38)
          :SLOPE 29.017
          :ORIENTATION 45.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (15.0 75.0 195.0 255.0)
          :BRAKING-CONSTRAINTS (333.535 116.465))

#S(REGION :EDGE-LIST (E-26 E-38 E-30 E-37)
          :SLOPE 29.017
          :ORIENTATION 315.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (285.0 345.0 105.0 165.0)
          :BRAKING-CONSTRAINTS (243.535 26.465))

#S(REGION :EDGE-LIST (E-25 E-37 E-29 E-40)
          :SLOPE 29.017
          :ORIENTATION 225.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (195.0 255.0 15.0 75.0)
          :BRAKING-CONSTRAINTS (153.535 296.465))

#S(REGION :EDGE-LIST (E-35 E-24 E-36 E-82)
          :SLOPE 24.376
          :ORIENTATION 135.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (80.0 190.0 260.0 10.0)
          :BRAKING-CONSTRAINTS (67.901 202.099))
```

```
#S(REGION :EDGE-LIST (E-34 E-23 E-35 E-81)
          :SLOPE 24.376
          :ORIENTATION 45.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (350.0 100.0 170.0 280.0)
          :BRAKING-CONSTRAINTS (337.901 112.099))

#S(REGION :EDGE-LIST (E-33 E-22 E-34 E-80)
          :SLOPE 24.376
          :ORIENTATION 315.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (260.0 10.0 80.0 190.0)
          :BRAKING-CONSTRAINTS (247.901 22.099))

#S(REGION :EDGE-LIST (E-21 E-33 E-79 E-36)
          :SLOPE 24.376
          :ORIENTATION 225.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (170.0 280.0 350.0 100.0)
          :BRAKING-CONSTRAINTS (157.901 292.099))

#S(REGION :EDGE-LIST (E-63 E-64 E-65 E-66)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (E-12 E-19 E-73 E-20)
          :SLOPE 18.166
          :ORIENTATION 90.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (5.0 175.0 185.0 355.0)
          :BRAKING-CONSTRAINTS (32.505 147.495))

#S(REGION :EDGE-LIST (E-11 E-18 E-72 E-19)
          :SLOPE 18.166
          :ORIENTATION 0.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (275.0 85.0 95.0 265.0)
          :BRAKING-CONSTRAINTS (302.505 57.495))
```

```
#S(REGION :EDGE-LIST (E-10 E-17 E-71 E-18)
          :SLOPE 18.166
          :ORIENTATION 270.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (185.0 355.0 5.0 175.0)
          :BRAKING-CONSTRAINTS (212.505 327.495))

#S(REGION :EDGE-LIST (E-9 E-20 E-74 E-17)
          :SLOPE 18.166
          :ORIENTATION 180.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS (95.0 265.0 275.0 85.0)
          :BRAKING-CONSTRAINTS (122.505 237.495))

#S(REGION :EDGE-LIST (E-4 E-16 E-8 E-15)
          :SLOPE 15.709
          :ORIENTATION 90.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (38.824 141.176))

#S(REGION :EDGE-LIST (E-3 E-15 E-7 E-14)
          :SLOPE 15.709
          :ORIENTATION 0.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (308.824 51.176))

#S(REGION :EDGE-LIST (E-2 E-14 E-6 E-13)
          :SLOPE 15.709
          :ORIENTATION 270.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (218.824 321.176))

#S(REGION :EDGE-LIST (E-1 E-13 E-5 E-16)
          :SLOPE 15.709
          :ORIENTATION 180.0
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS (128.  ; 231.176))
```

```
#S(REGION :EDGE-LIST (BE-8 E-21 BE-9)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-9 BE-10 E-S)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-10 E-24 BE-11)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-5 E-E BE-11)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-4 BE-5 E-23)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-7 BE-6 E-22)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)
```

```
#S(REGION :EDGE-LIST (BE-3 BE-4 BE-6 E-4)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-2 E-N BE-3 E-3)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-1 E-W BE-2 E-2)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

#S(REGION :EDGE-LIST (BE-8 BE-1 E-1 BE-7)
          :SLOPE 0.0
          :ORIENTATION NIL
          :SURFACE-MATERIAL GRAVEL-CLAY
          :SURFACE-CONDITION DRY
          :SURFACE-COVERING BRUSH
          :TYPE NIL
          :STABILITY-CONSTRAINTS NIL
          :BRAKING-CONSTRAINTS NIL)

;*******************************************************************************
```

APPENDIX C - VEHICLE STRUCTURES

```
;*****************************************************************************
;
; Vehicle List:
;
;*****************************************************************************

(M113-APC M966-ATC M813-CT)

;*****************************************************************************
;
; Structures:
;
;*****************************************************************************

#S(VEHICLE :NAME ARMORED-PERSONNEL-CARRIER
           :TYPE TRACKED
           :WEIGHT 24594.0
           :CENTER-OF-GRAVITY NIL
           :COASTING-SLOPE 10.0
           :CONTOUR-SLOPE 17.0
           :GRADIENT-SLOPE 31.0
           :STABILITY-SAFETY-MARGIN NIL)

#S(VEHICLE :NAME ARMORED-TOW-CARRIER
           :TYPE WHEELED
           :WEIGHT 7900.0
           :CENTER-OF-GRAVITY NIL
           :COASTING-SLOPE 5.0
           :CONTOUR-SLOPE 22.0
           :GRADIENT-SLOPE 31.0
           :STABILITY-SAFETY-MARGIN NIL)

#S(VEHICLE :NAME CARGO-TRUCK
           :TYPE WHEELED
           :WEIGHT 21020.0
           :CENTER-OF-GRAVITY NIL
           :COASTING-SLOPE 5.0
           :CONTOUR-SLOPE 17.0
           :GRADIENT-SLOPE 31.0
           :STABILITY-SAFETY-MARGIN NIL)

;*****************************************************************************
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Director of Research Administration, Code 012 Naval Postgraduate School Monterey, California 93943	1
4.	Robert B. McGhee, Code 52Mz Department of Computer Science Naval Postgraduate School Monterey, California 93943	10
5.	Neil C. Rowe, Code 52Rp Department of Computer Science Naval Postgraduate School Monterey, California 93943	10
6.	Michael J. Zyda, Code 52Zk Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
7.	C. Thomas Wu, Code 52Wq Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
8.	Harold M. Fredricksen, Code 53Fs Department of Mathematics Naval Postgraduate School Monterey, California 93943	1
9.	Edward B. Rockower, Code 55Rf Department of Operations Research Naval Postgraduate School Monterey, California 93943	1
10.	Major Ron S. Ross, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	50