NPSCS-91-006

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A232 224

Rule-Based Motion Coordination
For The Adaptive Suspension Vehicle
On Ternary-Type Terrain

S. H. Kwak
and
R. B. McGhee

December 1990

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, California 93943-5100

DTIC
ELECTE
FEB 26 1991
S B D

91 2 22 043

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared in conjunction with research funded by the Ohio State University Research Foundation.

Reproduction of all or part of this report is authorized.

Sehung Kwak
Adjunct Professor
of Computer Science

Reviewed by:

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

PAUL J. MARTO
Dean of Research

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) NPSCS-91-006	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (If applicable) CS	7a. NAME OF MONITORING ORGANIZATION Prof. Kenneth Waldron, D. of Mech. Eng. Ohio State University
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5100	7b. ADDRESS (City, State, and ZIP Code) 2075 Robinson Laboratory, 206 W 18th Ave. Columbus, Ohio 43210
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Ohio State Univ. Research Found.; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER RF Project #716520 & RF Purchase Order #496549
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Rule-Based Motion Coordination For the Adaptive Suspension Vehicle on Ternary-Type Terrain

12. PERSONAL AUTHOR(S)
S. H. Kwak and R. B. McGhee

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 10/88 TO 12/89	December 1990	177

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Robotics, Walking Machines, Adaptive Suspension Vehicle, Robot Motion Planning, Rule-based systems

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

This study investigates the utility of rule-based coordination of motion for ternary-type terrain locomotion by a hexapod walking machine. The ternary-type terrain considered is composed of permitted areas, forbidden areas, and ditch areas. The logic for generating motion coordination is written in Prolog while the simulation of the terrain and of the vehicle kinematics, as well as low-level on-board computer functions, are written in extended Common Lisp and Flavors. It is found that this approach, which utilizes multiple programming paradigms for programming motion coordination logic and simulation objects, results in code that is much easier to understand and modify than previous motion coordination programs written in Pascal. Thus, the code development effort and time are greatly reduced. The authors believe that both the methodology and the motion coordination logic presented in this report possess sufficient merit to justify full-scale physical testing in the Adaptive Suspension Vehicle at the Ohio State University.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Sehung Kwak	22b. TELEPHONE (Include Area Code) (408) 646-2168; 22c. OFFICE SYMBOL CS/Kw

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

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Rule-Based Motion Coordination For The Adaptive Suspension Vehicle On Ternary-Type Terrain

S.H. Kwak and R.B. McGhee

Naval Postgraduate School
Department of Computer Science (Code CS)
Monterey, CA 93943, U.S.A.

ABSTRACT

This study investigates the utility of rule-based coordination of motion for ternary-type terrain locomotion by a hexapod walking machine. The ternary-type terrain considered is composed of permitted areas, forbidden areas, and ditch areas. The logic for generating motion coordination is written in Prolog while the simulation of the terrain and of the vehicle kinematics, as well as low-level on-board computer functions, are written in extended Common Lisp and Flavors. It is found that this approach, which utilizes multiple programming paradigms for programming motion coordination logic and simulation objects, results in code that is much easier to understand and modify than previous motion coordination programs written in Pascal. Thus, the code development effort and time are greatly reduced. The authors believe that both the methodology and the motion coordination logic presented in this report possess sufficient merit to justify full-scale physical testing in the Adaptive Suspension Vehicle at the Ohio State University.

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

1. Introduction

The Adaptive Suspension Vehicle (ASV) is a large six-legged vehicle designed for outdoor operation in rough terrain. Limb motion coordination for the ASV is accomplished by an on-board computer network consisting of one PC-AT, eight Intel single-board computers, and two special purpose computers [1,2]. The software system is hierarchically organized with a clear distinction being made among an individual leg control level, a leg motion coordination level, and a body motion planning level [2,3]. Except for the two special purpose computers, the application software for the ASV is currently written almost entirely in Pascal. A custom designed real-time operating system, written mainly in PL/M, coordinates the functioning of all processes running on the various processors of the vehicle computer. The total ASV software system involves somewhat more than 150,000 lines of code [2,4].

An important feature of the ASV is its omni-directional motion capability [1,2] which gives it the general maneuverability characteristics of a helicopter. This behavior is achieved by providing the operator with a joystick with three major motion axes for control of vehicle forward velocity, lateral velocity, and turning velocity respectively [2]. The vehicle control computer accepts these commands and synthesizes a sequence of leg movements to produce the desired body behavior. It is assisted in this task by information from an optical terrain scanner which provides a map of terrain elevation in the immediate vicinity of the vehicle [5], and by force and position feedback from each leg.

Until now, nearly all outdoor experiments with the ASV have made use of a *tripod* gait in which legs are used in two sets of overlapping tripods [6,7]. This gait was chosen both for its relative simplicity and for its known optimality under high speed straight-line locomotion conditions [7,8,9]. However, the tripod gait is not well suited to extreme terrain situations in which a significant fraction of the area under a given leg may be unsatisfactory for load bearing due to the presence of rocks, holes, obstacles, soft soils, etc. In the latter case, simulation experiments [10 ,11], and initial indoor testing [12], indicate that on-line optimization of leg sequencing should give better results.

Gaits involving real-time optimization of stability or maneuverability in the presence of terrain constraints are often called *free* gaits to distinguish them from the periodic gaits used by walking machines and animals in less difficult circumstances [13,14]. Until now, all free gait studies have been performed

based on a binary terrain model which includes two types of terrain objects; i.e., obstacle and non-obstacle [10,11,15,16] . The size of the individual obstacles is restricted to be comparable to that of the feet of the ASV with the further assumption that the obstacles are randomly distributed on the terrain. In this study, in addition to the above small obstacles, a ditch obstacle, which is a large and structured obstacle whose size is comparable to that of the vehicle, is introduced as a third type. Therefore, the terrain model is ternary rather than binary, and contains both randomly distributed small obstacles and large ditch obstacles. Though the free gait motion coordinator developed in [16] performs well for binary terrain, in this report, a new motion coordinator, called the "Ditch Crossing Motion Coordinator" is additionally introduced to enhance ditch crossing capability. Thus, two motion coordinators coexist in the program, while, at one instance, only one of them is allowed to control the vehicle depending on terrain conditions. If the existence of a ditch is detected, the newly introduced motion coordinator takes over control of the vehicle until the ASV has crossed the ditch. After crossing the ditch, vehicle control is automatically transferred back to the Free Gait Motion Coordinator developed in [16] to handle small obstacles effectively.

Differing from all ASV experiments using an imperative language (Pascal) to encode stepping algorithms, this study uses Prolog, Flavors [18], and Lisp as in [16,17] because of the authors' belief that such a multiple programming paradigm is very well suited to the complex nature of the motion coordination problem for the ASV, involving a constant interaction among the on-line optimization logic, the vehicle and its internal objects, and the numerical routines. In what follows, Prolog is used to program the first part, and Flavors and Lisp are introduced to program the second and the third parts. This division of the motion coordination problem is suggested by the physiology of animals which utilize a brain, a physical body, and muscles to move themselves. Moreover, like the hierarchical organization existing among the these functional parts of animals, the program described in this report is also organized according to same type of hierarchy. Specifically, the top level of the program is the motion coordinator written in Prolog, while the second level is the Flavor objects which simulate a body, legs, and sensory organs. The numerical function routines written in Lisp perform all necessary calculations to move the legs and the body analogous to the action of muscles. The resulting code is remarkably easy to understand and modify because each programming language naturally provides the right programming paradigm for each division of

the program. Moreover, the resulting code is at least an order of magnitude shorter than the corresponding Pascal code for same reason. Consequently, this approach significantly reduces development time.

The remainder of this report first presents definitions used in this report, and a discussion of the mathematical-model used to simulate terrain and the ASV vehicle. This is followed by a description of the ditch crossing motion coordination rule-set, and the use of Prolog to realize both ditch crossing motion coordination and normal motion coordination. The report concludes with a discussion of the results of the investigation and suggestions for future research.

2. Definitions

In this report, a number of definitions are used as follows:

Definition 1: A *foothold* is a point on a segment of terrain, and can be assigned to a leg while the leg is in the air. When the foot of a leg is placed on the terrain, its assigned foothold becomes the *support point* of the leg. A foothold associated with a leg can be changed to a new one before the foothold becomes a support point [10].

Definition 2: The *support pattern* associated with a given set of leg support points is the convex hull of the vertical projections of all support points into a horizontal plane [13].

Definition 3: The magnitude of the *stability margin* at time t for an arbitrary support pattern is equal to the shortest distance from the vertical projection of the vehicle center of gravity to any point on the boundary of the support pattern. If the pattern is statically stable, the stability margin is positive. Otherwise, it is not defined [15].

Definition 4: A *working volume* is associated with each leg. This volume is a subset of three-dimensional space defined relative to the body and consists of the collection of points which can be reached by the foot of the given leg [11,19].

Definition 5: A *temporal kinematic margin* is associated with each foothold. At any instant, this margin is the time remaining until the associated leg would reach the boundary of its working volume if the foothold were used as a support point [15,19].

3. Vehicle and Terrain Model

While the vehicle model used in this study is based on the ASV, it represents only the major vehicle dimensions and components. Specifically, the cabin and the terrain scanner are omitted from the simulation model, while the geometries of the body and the legs are identical to those of the ASV. Therefore, the simulation model is represented by a simple six-faced box with each leg drawn as two line segments as shown in Figure 1. The exact vehicle dimensional data can be found in other literature [2,7]. Differing from the most previous simulation studies related to gaits and control of stepping [10,11,15,16,19], which simply ignore the overlapped portions of adjacent working volumes, in this study, the overlapped portions are taken into account during the foothold selection process in order to utilize the full kinematic capability of the vehicle.

The terrain adopted for this study is made up of terrain cells, and these individual cells are classified into two types of cells. One type, called a *permitted* cell, is able to support the body load when a leg steps on it. The other type, named a *forbidden* cell, is not usable because of unfavorable terrain conditions. Though this classification is complete with respect to individual terrain cells, there is a chance that a group of the forbidden cells can constitute a large structured obstacle instead of being randomly distributed. In this study, one type of structured large obstacle, a *ditch*, is considered because of its special shape, a long length and a relatively narrow width. Due to its shape, the most effective way to overcome a ditch is, if possible, crossing it instead of avoiding it by going around it. However, avoidance may be a better choice for other

types of large obstacles. This possibility is not studied in this report. Rather, the simulation terrain is ternary terrain which is composed of *permitted* cells, *forbidden* cells, and a *ditch*. A typical terrain example utilized in this study is shown in Figure 1. A cell with an "X" mark is a forbidden cell or a part of a ditch area while unmarked cells are permitted. A ditch is shown in the middle of the simulation terrain. Forbidden cells on this terrain can be designated either manually by an operator or automatically by using a random number generator with a threshold chosen to produce a specified ratio between permitted cells and forbidden cells [11].

The dimensions of each cell are one foot by one foot. This size is comparable to that of the feet of the ASV, and is larger than the resolution of the terrain scanner [5,12].

An overall block diagram of the program developed in this study is shown in Figure 2. This entire program is written for a Symbolics 3650 Lisp machine [11,20,21]. Each box shown is an object that is an instance of a Flavor [18] with the exception of the Free Gait Coordinator which is written in Symbolics Prolog [22]. Like the physical ASV which has nine major parts, namely, a body, a vision sensor, a cab, and six legs, the simulation object, "ASV" has correspondingly nine component objects, "Body", "Vision Sensor", "Joystick", and "Leg1" through "Leg6". These nine objects are linked to "ASV" through a *part* relation [16,23]. Each part has its sub-parts, and again is linked to them with a part relation. Differing from the nine major parts which have visible corresponding parts in the real ASV, the subparts of the simulation are not physically tangible, but are introduced because of their functionalities for program development. For example, the "Leg1" object, which is a part of the "ASV" object, has six subparts: "Leg1 Plan Machine", "Leg1 Control Machine", "Leg1 Executor", "Leg1 Contact Sensor", "Leg1 Foothold Finder", and "Leg1 TKM Calculator". Through the use of a *make-instance* function in an appropriate Flavor slot, the "Leg1" object binds all of these subparts into one group with the part relation [18]. In order to show the above relations among the objects in Figure 2, the six subpart objects are drawn under the "Leg1" object.

Besides the part relation, Figure 2 also shows the hierarchical control structure linking the simulation objects. Specifically, communication is restricted between objects in two adjacent levels by an assumption that upper levels have the right to access status information at lower levels, but the latter must

receive explicit commands from upper levels to update their internal states. For example, when "ASV", the vehicle object, needs "Leg1" to support its body, it sends a "Place" decision to "Leg1" and continuously monitors "Leg1" as to whether "Leg1" has begun to support the body or is in motion to try to reach a foothold. On receiving a "Place" decision from "ASV", "Leg1" sends the "Place" decisions to "Leg1 Plan Machine" while making observations of this machine. This type of message passing to and status observation from subordinates continues until the "Place" decision is accomplished. That is, when the foot of "Leg1" actually hits the ground, the contact sensor of "Leg1" detects the event and changes its internal state. The state change of "Leg1 Contact Sensor" is observed by "Leg1 Executor" and by "Leg1 Control Machine". In this way, the state change in the lowest level is propagated to higher levels until the touch down event arrives at "ASV". The detailed description of this control scheme can be found in other literature [16].

The joystick object simulates the physical three-axis joystick of the ASV through the use of six keys on the simulation computer keyboard to increment or decrement each of the three rates controlled by the joystick. These rates are *forward velocity*, *lateral velocity*, and *turn rate*, all in body coordinates. The altitude of the vehicle above the terrain and its orientation in roll and pitch relative to the terrain are automatically regulated using the algorithms described in [24].

While an elementary representation of the vision sensor is included in the program, as described in the above discussion of terrain, it is assumed that all forbidden cells and ditches have already been identified by prior terrain analysis. Of course this assumption does not represent a physical limitation of the ASV, but as made merely to allow this simulation to be focused on vehicle control, rather than on vision.

In addition to simplification of vision, this simulation also ignores leg mass in order to avoid the complexity of computing a center of gravity which moves with respect to the body. Moreover, all inertial forces are omitted from the simulation. That is, as in most previous simulation studies relating to gaits and control of stepping [8,9,10,11,16,19,25,26], only *static* stability is considered in this study. While this simplification would be serious in high speed locomotion, free gaits are most appropriate to low speed traversal of extremely difficult terrain, so the authors do not feel that this is a serious limitation on the applicability of the results of this investigation.

4. Ditch Crossing Motion Coordination

When the vision system detects the existence of a ditch, the vehicle operation mode is switched from the normal free gait mode [11,16] to the ditch crossing mode. In contrast to the normal free gait mode which performs on-line optimization of leg stepping under an environment with randomly distributed small obstacles [11,16], the coordinator in the ditch crossing mode controls the vehicle with a predetermined motion sequence in order to effectively overcome a ditch; i.e., a structured large obstacle. The ditch crossing mode is composed of two phases, the *preparation* phase and the *main* phase, and these two phases are sequentially executed. The preparation phase provides a transition period from the normal free gait mode to the ditch crossing mode, and the main phase performs the actual ditch crossing action.

4.1 Preparation Phase

The preparation phase is composed of a sequence of nine states, and these nine states are grouped into two cycles which are named *Cycle* 1 and *Cycle* 2. The first cycle, *Cycle* 1, consists of six states and takes care of a transition from the normal mode to the ditch crossing mode. During the execution of the first cycle, the body attitude is modified to be suitable to cross a ditch. The second cycle, *Cycle* 2, consists of three states and causes the vehicle to have the correct leg configuration for the ditch crossing operation in the main phase. Thus, during the execution of *Cycle* 2, the body is not moved at all. The graphical representation of the preparation phase is shown in Figure 3.

The six states in *Cycle* 1 are *Place Legs in the Air*, *Back Middle Legs*, *Forward Rear Legs*, *Forward Middle Legs*, *Forward Front Legs*, and *Lift Middle Legs and Move*. Body movement is involved only in the last state. The first state, *Place Legs in the Air*, represents a simple action; i.e., leg placing, but the rest of the states represent at least two sequential actions. For example, during the *Forward Middle Legs* state, the middle legs are lifted from the ground and placed on the ground using one of the closest footholds to the front end of the working volumes of the middle legs. Similarly, in the *Back Middle Legs* state, the middle legs are lifted and placed at one of the closest footholds to the back end of the working volumes of the middle legs.

The first cycle of the preparation phase begins with the *Place Legs in the Air* state. During this state, all the legs in the air are placed on the ground without any body movement. At most three legs will be placed in this state because at least three legs must support the body at all times to maintain the stability of the vehicle. When a leg is placed in this state, one of the closest footholds to the front end of the working volume of each leg is selected as a stepping position on the ground. Thus, the newly placed legs have larger *temporal kinematic margins* (TKMs) than the legs already on the ground.

At the end of the first state, all six legs are on the ground. Thus, the middle legs can be used for any purpose because the front and the rear four legs are sufficient to make the vehicle stable as long as these four legs are within their kinematic limits. Therefore, the middle legs are used to provide maximum TKMs for the front and the rear legs in the following four states.

The *Back Middle Legs* state is the first state of the sequence to maximize TKMs of the front and the rear legs. The middle legs are lifted and placed at the back ends of their working volumes. At the completion of this state, the middle legs are placed behind the center of the gravity of the vehicle. Thus, the vehicle can maintain its stability with the front and the middle legs alone, and the rear legs can be lifted.

In the *Forward Rear Legs* state, the rear legs are lifted and placed at the front end of the working volume of the rear legs. Thus, both rear legs will have maximum TKMs at the end of this state. Though the rear leg lifting actions are inherently safe, before lifting one of the rear legs the vehicle stability is checked to ensure that the vehicle is stable without the rear leg. If the vehicle is not stable, the leg is not lifted from the ground, and the ditch crossing operation will be halted. If it is stable without the leg, the leg is lifted. After the first rear leg is lifted safely, the same test is performed on the other rear leg before it is lifted. This type of test is always performed before lifting any leg from the ground during the ditch crossing operation in order to ensure safe operation. At the end of the *Forward Rear Legs* state, both of the rear legs are placed. Again, the middle legs become redundant for the vehicle stability.

In the *Forward Middle Legs* state, the middle legs are lifted and placed at the front end of the working volumes of the middle legs. Because the new support points of the middle legs are ahead of the center of the gravity of the vehicle, the vehicle now can be safely supported by the middle and the rear legs. Thus, the front legs can be lifted from the ground without harming the stability of the vehicle.

In the *Forward Front Legs* state, the front legs are lifted and placed at the edge of the vehicle side of the ditch. This has to be done because the new support points of the front legs will be the last ones on the vehicle side of the ditch. This requirement is not hard to meet since, as long as the edge of the ditch is included in the working volume of the front legs, the front legs can always be put in the right position. However, the opposite side of the edge of the ditch should not be included in the working volume in order to prevent its being used as possible footholds for the front legs. Therefore, there is a range of the vehicle locations with respect to the near edge of the ditch so that the vehicle can select the right stepping positions for the front legs. This range is shown in the following equation:

$$P + \frac{1}{2}L - DW < DCIR < P + \frac{1}{2}L \qquad \text{.....} \qquad (1)$$

where DCIR : Ditch Crossing Initiation Range with respect to gravity center of the vehicle
P : Pitch between adjacent legs
L : Longitudinal length of working volume for each legs
DW : Ditch Width.

To understand this relationship, it should be recognized that the distance from the vehicle's center to the front ends of the working volumes of the front legs is the sum of the pitch between the middle and the front legs and half of the longitudinal length of the working volume of the front legs. Thus, the meaning of Eq. (1) is that when the ditch crossing operation is initiated, the front ends of the working volumes of the front legs should be positioned between the near edge and the far edge of the ditch. Evidently, if ditch crossing is initiated anywhere in the above range, the ditch crossing operation will not be hampered since the body attitude and the leg stepping positions are corrected during the preparation phase. It should be also noted, however, that if DW is less than 3 ft, the vision system does not have to detect the existence of the ditch at all. The normal plan developed in [16] is capable of handling such ditch width without any problem.

At the completion of the *Forward Front Legs* state, the front legs will be placed on the vehicle side edge of the ditch. Again, the middle legs become redundant.

In the *Lift Middle Legs and Move* state, the middle legs are lifted and the body is moved into the ditch area until at least one of the supporting legs (the front and the rear legs) reaches its kinematic limit. If too many obstacles have not interfered with the operations of the previous four states, there will be a high probability for the front legs to reach their kinematic limits first because footholds in the front most portion of the working volumes of the front legs may be excluded by the location of the ditch. Thus, at the end of this state, the vehicle body is fully pushed into the ditch area under the constraints of the current leg configuration, and its movement is stopped. At this point, although the body position and the front leg positions are right for the ditch crossing, the rear leg positions are not appropriate because they are already near their kinematic limits. Thus, further body movement is very limited or impossible depending on the kinematic margins of the rear legs. Even though the opposite side of the edge may not be reachable by the front legs at the end of the *Lift Middle Legs and Move* state, the vehicle can cross the ditch if it moves further into the ditch area by eliminating the kinematic problem of the rear legs and if the ditch width is narrower than the vehicle ditch crossing capability. In the second cycle of the preparation phase, the kinematic limits of the rear legs are eliminated.

The second cycle of the preparation phase has three states and starts with the *Back Middle Legs* state. No body movement is involved in the second cycle, but the stepping positions of the rear and the middle legs are rearranged.

In the first state, the middle legs, which are redundant to make the vehicle stable, are lifted and placed near the back end of their working volumes. Because the new support points of the middle legs are behind the center of the gravity of the vehicle, the rear legs can be lifted without harming the vehicle stability.

In the *Forward Rear Legs* state, the rear legs are lifted and placed near the front end of their working volumes. Thus, the rear legs obtain maximum kinematic margins. At this point, though the rear legs provide maximum body movement potential, the middle legs prohibit further body movement.

In the *Forward Middle Legs* state, the middle leg kinematic problem is eliminated. The middle legs are lifted and placed at the front end of the working volume of the middle legs. Therefore, both the middle and the rear legs have their maximum kinematic margins, while the body is completely pushed into

the ditch area. In contrast to the middle and the rear legs, the front legs should be at their kinematic limits because the front legs stepping positions have not changed since they were at their kinematic limits at the end of *Cycle 1*. If the front legs are now lifted, vehicle body movement can be resumed. This will be the first action of the following phase, the *Main Phase*.

In summary, at the end of the preparation phase, the body is fully moved into the ditch area within the limits of the stability of the vehicle and the kinematics of the legs. The middle legs and the rear legs are fully forward to enhance the ditch crossing capability, and the front legs are ready to be lifted from the ground. This preparation allows the vehicle to cross a wider ditch than the longitudinal length of the vehicle legs' working volumes. Though the other side of the ditch is not included in the front legs' working volumes at the end of the preparation phase, the body can move forward as long as the vehicle's stability is maintained and the leg kinematic limits are not reached. That is, if the other side of the ditch is included in the front leg's working volume before the vehicle becomes unstable and before the other legs reach their kinematic limits, the other side is reachable by the front legs. If the front legs can be placed on the other side within the vehicle's kinematic and stability limitations, then the vehicle can cross the ditch because the ASV legs' working volumes are identical and because the pitches between the front legs and the middle legs and between the middle legs and the rear legs are the same.

As a result of the above arguments, the maximum ditch width can be crossed by the ASV, which has identical working volumes for all legs and the equal pitches between the front and the middle legs and between the middle and the rear legs, is determined by both the pitch length and the length of the longitudinal working volumes of the legs. Specifically, the maximum ditch width can be crossed by the ASV is given by:

$$MDW = P + \frac{1}{2}L - SM - SDE \qquad \cdots\cdots \qquad (2)$$

where
- MDW : Maximum Ditch Width
- P : Pitch between adjacent legs
- L : Longitudinal length of working volume
- SM : Safety Margin
- SDE : Search Digitization Effect.

As can be seen, the maximum ditch width (MDW) is calculated by adding the pitch between adjacent legs and the half of the longitudinal length of the working volumes of the legs, and then by subtracting the safety margin (SM) and the search digitization effect (SDE). The safety margin is a prescribed margin ensuring safe operation of the vehicle. The search digitization effect is an artifact of the foothold search process resulting from a one foot by one foot grid search. With the dimensions of the simulation model discussed in [2,7], the MDW becomes 8.5 ft when the SM and the SDE are 0.5 ft, respectively.

4.2 Main Phase

The *Main Phase* is composed of three cycles The first and the third cycles are composed of three states each, while the second cycle contains only one state. The first cycle in the main phase in the program is named *Cycle 3* to show continuation from the preparation cycles. Consequently, *Cycle 1* and *Cycle 2* belong to the *Preparation Phase*, and *Cycle 3* through *Cycle 5* belong to the *Main Phase*. A graphical representation of the *Main Phase* is shown in Figure 4.

The first cycle, *Cycle 3* is composed of three states, *Move Forward Front Legs*, *Move Back Middle Legs*, and *Move Forward Rear Legs*. During this cycle, the front legs cross the ditch, and the rear legs are prepared to replace the middle legs which will cross the ditch in the following cycle. In this cycle, the vehicle body is allowed to move forward whenever possible. Therefore, all the state names are pre-fixed with "Move", and each state is composed of three actions, leg lifting, body movement, and leg placement.

In the *Move Forward Front Legs* state, first, the front legs are lifted while the body is not moved. As soon as both front legs are lifted from the ground, the second action is performed, which is a forward body movement. This body movement is sustained until the middle legs limit this movement because the middle leg positions with respect to the center of the vehicle gravity determine the stability margin when only the middle and the rear legs support the body. Though the middle legs can kinematically move behind the center of gravity, they should be stopped in front of the center of gravity to maintain the safety stability margin.

When the body movement is stopped with the completion of the second action of the current state, the opposite side of the ditch will be included in the working volumes of the front legs if the width of the

ditch is narrower than MDW. Thus, the third action of the *Move Forward Front Legs* state follows, in which the front legs are placed on the opposite side of the ditch. Thus, the middle legs become redundant for the vehicle stability.

In the second state of *Cycle 3*, the *Move Back Middle Legs* state, the middle legs are lifted from the ground. The vehicle body movement is resumed because the movement is restricted by the middle legs to maintain the vehicle stability margin. The body movement is continued until any one of the supporting legs meets its kinematic limit. Specifically, the body will be moved until one or both rear legs reach its or their kinematic limits because the kinematic margins of the rear legs have been used to move the body in the previous state, but those of the front legs have been just maximized in the previous state, the *Move Forward Front Legs* state of *Cycle 3*. This effect can be easily seen in the second drawing of *Cycle 3:State 2* in Figure 4, and "Rear Legs" written on the top of the second drawing shows the termination condition of the current body movement. When the kinematic limits of the rear legs stop the body movement, the third action of the current state is performed, which is to place the middle legs at the back end of their working volumes. Because the middle legs are placed behind the center of gravity, the rear legs can be lifted from the ground while the front and the middle legs stably support the body.

In the *Move Forward Rear Legs* state, which is the third state of *Cyclc3*, first, the rear legs are lifted from the ground. As soon as the rear legs are lifted, the body movement is resumed. However, the body movement is immediately blocked by the middle legs which have placed back in the previous state. Thus, the third action of the current state, which places the rear legs at the front end of their working volumes, is immediately started. Consequently, very little body movement is involved in this state, but the rear legs gain large TKMs so that the body can be moved further in the next cycle. Therefore, though the other side of the ditch may not reachable by the middle legs under the current body position, the new body movement, which will be performed in the next cycle, will make this possible.

The second cycle of the main phase, *Cycle 4*, is composed of one state, the *Move Forward Middle Legs* state. In this cycle, the middle legs will be moved to the other side of the ditch. The first action is lifting the middle legs so that the body movement can be resumed. This movement will last as long as the

front legs have positive TKMs because the TKMs of the front legs have already been partially consumed in the previous cycle. This is shown in the second drawing of *Cycle 4:State 1* in Figure 4.

When the body movement is stopped, the other side of the ditch is reachable by the middle legs because the geometries of the front and the middle legs are identical and because the working volumes of the front and the middle legs overlap slightly at the rear end of the former volume and the front end of the latter volume. As soon as the middle legs are positioned on the other side of the ditch, this cycle is terminated.

The last cycle, *Cycle 5*, which is the third cycle of the main phase, takes care of the ditch crossing action of the rear legs. This cycle is composed of the three states, the *Move Forward Front Legs* state, the *Move Back Middle Legs* state, and the *Move Forward Rear Legs* state. These three states are the same that of *Cycle 3*. This is not a coincidence, but an expected consequence of the geometrical symmetry of the front and the rear legs.

In the *Move Forward Front Legs* states, the kinematic problems of the front legs which block further body movement are relieved because the first action of this state is to lift the front legs from the ground. Thus, the body movement, which is the second action of this state, is resumed, and is terminated by the positions of the middle legs with respect to the body because the positions of the middle legs determine the stability of the vehicle. When this state is terminated, the front legs are placed on the ground.

In the *Move Back Middle Legs* state, the body movement is resumed as soon as the middle legs are lifted from the ground. This movement will last until the kinematic limits of the rear legs are reached. When this condition is met, the body movement is stopped and the middle legs are placed as far backward as possible so that the middle legs can support the body together with the front legs. Consequently, the middle legs will be placed at the edge of the ditch because this edge is in the working volumes of the middle legs.

The *Move Forward Rear Legs* state, which is the last state in the last cycle, causes the rear legs to cross the ditch. The first action is to lift the rear legs from the ground. Body movement is then resumed and continued until the other side of the ditch is reachable by the rear legs. Again, this will be accomplished because the working volumes of the middle and the rear legs slightly overlap. Finally, all the legs are across the ditch. Thus, the ditch crossing operation has accomplished and the operational mode is

switched back to the normal mode, and the Free Gait Motion Coordinator [16] regains control of the vehicle.

5. Program Implementation

The top level Ternary Terrain Motion Coordinator is written in Symbolics Prolog because of its easy translation characteristics from natural language to a computer program, and because of its straightforward interface to Symbolics Lisp language in which the rest of the program is written. The Prolog program is listed in Figure 5. It is composed of three functional groups of predicates. The first group controls the flow of the whole program, while the second does logic processing which generates commands for the vehicle body and legs. The last group is responsible for bridging between the program written in Prolog and the robot program in Flavor objects. This is accomplished through the Lisp function call facility provided by Symbolics Prolog. Specifically, anything following the "is" predicate in a Prolog clause may be either a Prolog arithmetic function or the name of a Lisp function [22]. If a Lisp function name follows the "is" predicate, it is evaluated according to its definition inside the Lisp environment. In the program of Figure 5, arguments following "is" predicates are names of Lisp functions, and make connections to the Lisp environment. A returned value resulting from a Lisp function call may be used to instantiate a variable preceding the "is" Prolog predicate or test whether the returned value matches a value preceding the "is" predicate. In the former case, the subgoal "is" always succeeds, but the latter case, only when two values agree does the "is" subgoal succeed. In the program, only the former case is used. The Lisp portion of the program is listed in the appendix attached at the back of this report.

The Prolog program is started by typing "robot" on the computer console. The *robot* clause is the first line of the program. After the initialization process is done, it makes the *loop* clause repeat. Thus, it determines the flow of the whole program.

The *loop* clause is composed of three subgoals, *get_command*, *plan*, and *execute*. This shows the flow of the program execution for each loop. Based on the input command from the joystick, motion is

planned, and the planned motion is executed. Then, the executed motion is drawn on the screen by the *draw_robot* clause which actually calls a corresponding Lisp function, *graphical_display*.

The *plan* subgoal of the *loop* clause has two alternatives, *free_gaits_motion_coordination_plan* and *ditch_crossing_motion_coordination_plan*. In the Prolog program, the *ditch_mode* subgoal is tested first because *ditch_crossing_motion_coordination_plan* deals with a more specific case than the other. If the *ditch_mode* subgoal succeeds, then *ditch_crossing_motion_coordination_plan* is executed. If not, then *free_gaits_motion_coordination_plan* is executed. The *free_gaits_motion_coordination_plan* clause is composed of *update_robot_state*, *check_tkm_limit*, *leg_plan*, *body_plan*, and *generate_decision* subgoals. The first and the second subgoals update the state and the body position of the vehicle and check kinematic problems of the legs. The third subgoal, *leg_plan*, performs on-line optimization for leg coordination using the free gait strategy [11,16]. Based on the leg plan, the fourth subgoal, *body_plan*, plans the body movement to enhance the vehicle stability. Finally, the *generate_decision*, subgoal sends decisions to the "ASV" robot flavor object. A detailed description of the *free_gaits_motion_coordination_plan* can be found in other literature [11,16].

The *ditch_crossing_motion_coordination_plan* and the related clauses implement the ditch crossing coordination discussed in the previous section. There are two *ditch_crossing_motion_coordination_plan* clauses in the program, and the first clause checks the termination condition while the second clause performs the ditch crossing planning. If the ditch crossing activity is not completed, then the first clause fails, and the second *ditch_crossing_motion_coordination_plan* clause is executed. Thus, the *cycle_planner* clause is called into an action.

The *cycle_planner* clause and related clauses follow the above two *ditch_plan* clauses. This group of clauses is named "Cycle Planner". The first clause of the "Cycle Planner" group, *ditch_plan_done*, checks the completion of the ditch crossing plan. The two *cycle_planner* clauses take care of ditch plan cycle changes from *cycle 1* to *cycle 5* by increasing the cycle number whenever one cycle is completed. Therefore, the *ditch_plan_done* clause succeeds as soon as *cycle 5* is finished because the cycle number becomes 6 immediately after the completion of *Cycle 5*. The last subgoal of *ditch_plan_done*, is

idle_cycle, which is a dummy plan without any leg planning. It is introduced to fill the gap for the transition between cycles.

The subgoals used in the *cycle_planner* clauses, *one_cycle_done* and *plan_cycle*, are grouped in the following part of the program, which is called the "Plan Cycle Dispatcher" group. The *one_cycle_done* clause checks the termination condition of one plan cycle, and the five *plan_cycle* clauses execute the appropriate ditch plan cycles based on the plan cycle number which is given through the *plan_cycle* fact in the Prolog data base.

The *ditch_plan_cycle* clauses, *ditch_plan_cycle1* through *ditch_plan_cycle5*, form another group called "Cycles" in the program following the "Plan Cycle Dispatcher" group. The first subgroup, *ditch_plan_cycle1*, has seven clauses. The first clause of this subgroup takes care of the initial state transition, and the rest of them represent the six states in *cycle 1* discussed in the previous section. Specifically, the first clause retracts *plan_state(start)*, which is a cycle starting fact, from the Prolog data base and asserts a new fact, *plan_state(place_legs_in_the_air)*, which is the name of the first state. After changing the Prolog data base, the first clause executes the *place_legs_in_the_air* subgoal, which performs the *place legs in the air* state in *Cycle 1*. When the *place_legs_in_the_air* subgoal is executed, the first clause provides the next state information for the subgoal so that when the current subgoal is completed the correct information about the succeeding state is asserted in the data base. The second clause through the seventh clause sequentially represent six states in *cycle 1*. Thus, these ordered clauses represent the sequence of state transitions among the six states. When the last clause calls the *lift_middle_legs_and_move* subgoal, *one_plan_cycle_done* is given instead of the name of the next state to assert the cycle termination fact in the data base. This structure is repeated for the rest subgroups of the "Cycles" group, *ditch_plan_cycle2* through *ditch_plan_cycle5*.

The following group called "States" is composed of 10 different subgroups, and each subgroup is composed of two clauses. These clauses accept information about the next state so that the next state information is asserted when the current state is completed. However, only the first clause, which takes care of its state transition, utilizes the next state information. The second clause ignores the next state information and executes a subgoal whose name is its clause head name pre-fixed with "do_". Additionally,

both clauses of each subgroup determine the body movement by executing either the *stop* or the *move* subgoal depending on the needs of each state described in the previous section.

The "State Executors" clause group follows the "States" clause group. This clause group is composed of 11 subgroups of clauses. Among them, 10 subgroups are responsible for execution of 10 states in the "States" group, while the eleventh subgroup takes care of the body movement, such as *move*, *stop*, *clear_move_memory*, and *move_done*. Therefore, except for the last subgroup, each subgroup shows a sequence of actions within a state, which are described in the previous section. For example, the *do_back_middle_legs* clause subgroup, which is the first subgroup of the "State Executors" group, is started with three major clauses. The first clause, *back_middle_legs_done*, tests the state termination condition, and the second and the third clauses perform a sequence of actions, which are lifting the middle legs and then placing them at the back side of their reachable areas. The second clause tests whether both middle legs are lifted by executing the subgoal, *all_middle_legs_lifted*. Initially, this test should fail. Thus, the third clause is executed. After the third clause is executed twice, the both middle legs will have been lifted from the ground because the *lift_middle_legs* subgoal causes one middle leg to be lifted from the ground at a time. In the *do_back_middle_legs* subgroup, there are two *lift_middle_legs* clauses. The first clause performs the leg lifting action by selecting one middle leg and then causing it to be lifted from the ground, while the second clause performs a default action by always succeeding. Only when the both middle legs are lifted from the ground, is the middle leg placement executed. Specifically, when the *all_middle_legs_lifted* subgoal in the second clause of this subgroup is satisfied, the *place_middle_legs_back* subgoal is executed. If the middle legs are placed on the ground again, then the *back_middle_legs_done* clause, which is the first clause of the current subgroup, succeeds because the *all_middle_legs_lifted* subgoal has been satisfied by the *middle_legs(lifted)* fact which was asserted when middle legs were lifted from the ground, and because the *all_middle_legs_placed* subgoal is now satisfied. Before completing the *back_middle_legs_done* clause, the first clause of the current subgroup, the *clear_middle_lifted_memory* and the *clear_move_memory* subgoals clear residual facts generated during execution of the *do_back_middle_legs* clauses in order not to interfere with the execution of the following "State Executions" subgroup clauses.

Rest of the subgroups in the "State Executors" group have the exactly same structure that of the *do_back_middle_legs* clause subgroup. Specifically, one state termination clause is followed by two state execution clauses which are pre-fixed with "do_" and related clauses which support these leading three clauses. If the related clauses are already available, they are not duplicated by adding them in the subgroup.

The only exception to the above structure is the fifth subgroup, the *do_lift_middle_legs* clause subgroup. This subgroup is composed of one clause, and there is no clause to test the state termination condition. The time required to complete the middle leg lifting action in the *lift_middle_legs_and_move* state, which is the only state that utilizes the *do_lift_middle_legs* clause, is considerably shorter than that needed to complete the body movement in the state. Thus, the leg lifting action is always guaranteed before the current state is terminated.

The last group of clauses is named "Plan Libraries". These clauses are used by both *ditch_crossing_motion_coordination_plan* and *free_gaits_motion_coordination_plan*. This group is composed of two subgroups, *body_plan* and *generate_decision*. The latter subgroup sends planned leg motions through decisions to the robot, "ASV", which is a flavor object. It sends them one by one until all the decisions in the Prolog data base are exhausted. The former subgroup takes care of body movement by executing *speed_plan* and *trajectory_plan*. The *speed_plan* clauses control the speed of body movement and the *trajectory_plan* clauses modify body movement trajectory in order to increase the stability margin of the vehicle using a "push" operation which causes the gravity center of the vehicle to move away from the boundary of the current supporting pattern [16].

6. Discussion

Performance tests were carried out for various terrain conditions by making the ASV follow a prescribed standard trajectory. The standard trajectory is a straight line across the model terrain from one side to the other side while crossing a ditch oriented perpendicular to the direction of vehicle motion. No failures to complete the standard trajectory were observed for any terrain containing up to a 8 ft width ditch if no randomly distributed forbidden cells were included in the terrain. However, when forbidden cells were

added to the terrain with a 8 ft width ditch, the performance was severely degraded. If the randomly distributed forbidden cells occupied 30 percent or more of the area of the non-ditch portion of the whole simulation terrain, the ASV always failed to complete the standard trajectory. Specifically, the ditch crossing operation was halted because the random obstacles on the ground prevented the ASV from using the most favorable stepping positions near the ditch. However, when the width of a ditch was reduced to 7 ft, no failures in ditch crossing operations were observed. Rather, the capability to overcome randomly distributed forbidden cells became the bottle neck which determined whether the ASV could complete the standard trajectory or not. Overall, when less than 70 percent of the total terrain cells were the forbidden cells, the program made the ASV follow the standard trajectory without great difficulties.

One of the advantages of using object-oriented programming for the ASV object and its subobjects was the easy extension to a new ASV with additional functionality required for the ditch crossing maneuvers. Specifically, this was accomplished by using the inheritance mechanism provided by Flavors. The original ASV was an instance of "robot" class, and the new extended ASV is an instance of "ditch-robot" class. The latter class is defined as a subclass of the former class. Thus, the entire functionality of the "robot" class became available to the "ditch-robot" class through the inheritance mechanism. The newly required capabilities were added to the "ditch-robot" class using "defmethod" which defines the functionality of a class in Flavors. The result was remarkable. The additional code written for the new "ditch-robot" class was less than 10% of the size of the original "robot" class, and roughly more than 95% of the original code was reused.

One of advantages of rule-based control of motion coordination is the ease of extension of coordination logic resulting from the fact that individual rules or a group of rules define an independent piece of behavior. Instead of rewriting all the code related to motion coordination, the new ditch crossing coordinator was simply added to the original Prolog code. In order to accept the new coordinator, the original *plan* Prolog goal in [16] was subdivided into two *plan* subgoals, *free_gaits_motion_coordination_plan* and *ditch_crossing_motion_coordination_plan*. The old *plan* code in [16] was merely renamed as *free_gaits_motion_coordination_plan* without any further modification, and the new *ditch_crossing_motion_coordination_plan* code was simply added. If the original motion coordinator logic had been imbedded

in the "ASV" robot code because only one programming paradigm, such as an imperative paradigm, had been utilized to program the work in [15], the extension to a ditch crossing capability in the "ASV" robot code would have been a very difficult and very time consuming task.

Overall, the development and coding of the new extended "ASV" and motion coordinator clearly manifested the advantages of the use of multiple programming paradigms to program a complex robot motion coordination function which constantly performs on-line optimization like a human or an animal coordinating his or its motion based on sensory information and learned experiences. Rule-based programming to express logic, object-oriented programming to simulate physical and functional objects, and a numerical processing library written in a functional or imperative language to implement mathematics and physics needed for simulation are very naturally divided components to simulate a complex system, such as that treated in this report.

One of major complains about programs using Artificial Intelligence techniques and languages is slow execution speed in on-line computing applications. In this study reported here, the most prominently visible candidate to be blamed for slow execution speed is Prolog code. However, the execution speed of Symbolics Prolog on a Symbolics Lisp machine is not so slow as might be expected. It is only slightly slower than that of Symbolics Lisp or Flavors. However, the execution speed of Prolog implementations on other machines are usually considerably slower than those of non-Prolog implementations. One solution for slow Prolog execution speed may be to use a special Prolog processor, such a Xenologic X-1 [27], to execute Prolog code. The other solution is to convert the Prolog code to an other language, such as Lisp. The latter approach was actually adopted to test the correctness of the program developed herein, using a TI Explorer machine because this solution is readily applicable without great modification to the interface between Prolog and Lisp codes, and because a TI Explorer machine was conveniently available to the authors in an office environment. Though the speed gain in program execution is little over that expected with Symbolics Prolog, this approach potentially makes a much wider variety of computing hardware suitable to execute the motion coordination program developed here. Moreover, this approach may provide another advantage in near future since advances in microprocessors based on RISC or CISC architecture [28] will, with respect to Lisp execution speed, soon equal or outperform Lisp machines [20].

Already, with respect to execution speed alone, SPARC-based Sun workstations narrow the large gap previously existing between a Lisp machine and a conventional machine running the Lisp language. Therefore, rather than running a slow Prolog program on a conventional machine, automatic conversion from Prolog to Lisp after a development phase could become an effective way to achieve markedly better performance if the program were to be tested on the physical ASV walking machine.

7. Summary and Recommendation

The main purpose of this study was to demonstrate the value of a multiple programming paradigm approach in the development of software for motion coordination for the ASV walking machine. An important secondary goal was extending the work in [16] so that the ASV can cross a ditch without any assistance from a human operator. Thus, the terrain handling capability of the ASV under program control was extended from binary-type terrain to ternary-type terrain for the first time. The third goal was to take into consideration the overlapping working volumes of the legs of the ASV in order to utilize the full kinematic capability that the vehicle geometry can give. This later factor made a direct contribution in widening the maximum ditch width (MDW) that the vehicle can cross.

The approach adopting multiple programming paradigms for motion coordination, which was proposed in [11] and implemented in [16], again exhibited its power. First of all, it forced a well-organized and functionally clean abstraction hierarchy for a complex and ill-defined problem. Secondly, it considerably reduced development time and effort. The program development associated with this report could have been a major undertaking if a single programming paradigm had been utilized. Instead, as described in the preceding text, with the approach taken here most of the program in [16] is reused, while only small amount of code is additionally written.

The code translation from Prolog to Lisp was possible because the Prolog code used herein was utilized as a simple rule-based system. This success of this translation further justifies the usage of Prolog as one of the languages in the multiple paradigm environment because it could allow much wider varieties

of computing hardware to execute the motion coordination program developed. Moreover, this actually made the program execution somewhat faster than that of the program with the untranslated Prolog code.

Among studies remaining to be conducted are inclusion of vehicle inertia in the simulation, effects of leg motion on the location of the vehicle center of gravity, and a better simulation of the vision system. Such a study would be appropriate to a later phase of this research along with an investigation of further changes to the Prolog rule set to enable the ASV to climb over large obstacles or to go around them if this is not possible.

References

[1] McGhee, R. B., "Computer Coordination of Motion for Omni-Directional Hexapod Walking Machines," *Advanced Robotics*, Vol. 1, No. 2, pp. 91-99, June 1986.

[2] Bihari, T. E., Walliser, T. M., and Patterson, M. R., "Controlling the Adaptive Suspension Vehicle," *IEEE Computer Magazine*, Vol. 22, No. 6, pp. 59-65, June 1989.

[3] McGhee R. B., Orin, D. E., Pugh, D. R., and Patterson, M. R., "A Hierarchically-Structured System for Computer Control of a Hexapod Walking Machine," *Theory and Practice of Robots and Manipulators*, pp. 375-381, ed. by A. Morecki et al, Hermes Publishing, 1985.

[4] Schwan, K., Bihari, T., Weide, B. W., and Taulbee, G., "High-Performance Operating System Primitives for Robots and Real-Time Control Systems," *ACM Transactions on Computer Systems*, Vol. 5, No. 3, pp. 189-231, August 1987.

[5] Kau, C. C., Olson, K. W., Ribble, E. A., and Klein, C. A., "Design and Implementation of a Vision Processing System for a Walking Machine," *IEEE Trans. on Industrial Electronics*, Vol. 36, No. 1, pp. 25-33, February 1989.

[6] Klein, C. A., Olson, K. W., and Pugh, D. R., "Use of Force and Attitude Sensors for Locomotion of Legged Vehicle over Irregular Terrain," *International Journal of Robotics Research*, Vol. 2, No. 2, pp. 3-17, Summer 1983.

[7] Song, S. M., and Waldron, K. J., *Machines that Walk: The Adaptive Suspension Vehicle*, MIT Press, Cambridge, Massachusetts, 1989.

[8] Bessonov, A. P. and Umnov, N. V., "The Analysis of Gaits in Six-Legged Vehicles According to Their Static Stability," *Proceedings of CISM-IFTOMM Symposium on Theory and Practice of Robots and Manipulators*, Udine, Italy, September 1973.

[9] McGhee, R. B., "Robot Locomotion," in *Neural Control of Locomotion*, pp. 237-264, ed. by R.R. Herman, et al., Plenum Press, New York, 1976.

[10] McGhee, R. B. and Iswandhi, G. I., "Adaptive Locomotion of a Multilegged Robot over Rough Terrain," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-9, No. 4, pp. 176-182, April 1979.

[11] Kwak, S. H., *A Computer Simulation Study of a Free Gait Motion Coordination Algorithm for Rough-Terrain Locomotion by a Hexapod Walking Machine*, Ph. D. dissertation, The Ohio State University, Columbus, Ohio, 1986.

[12] Pugh, D. R., et al., "Technical Description of the Adaptive Suspension Vehicle", *International Journal of Robotics Research*, Vol. 9, No. 2, April 1990.

[13] McGhee, R. B., "Walking Machines," *Advances in Automation and Robotics*, pp. 259-284, ed. by G. N. Saridis, Jai Press, Inc., 1985.

[14] Pearson, K. G. and Franklin, R., "Characteristics of Leg Movements and Patterns of Coordination in Locusts Walking on Rough Terrain," *International Journal of Robotics Research*, Vol. 3., No. 2, Summer 1984.

[15] Kwak, S. H., *A Simulation Study of Free-Gait Algorithm for Omni-Directional Control of Hexapod Walking Machines*, M. S. thesis, The Ohio State University, Columbus, Ohio, 1984.

[16] Kwak, S. H. and McGhee, R. B., "Rule-Based Motion Coordination For a Hexapod Walking Machine," *Advanced Robotics*, Vol. 4, No. 3, pp. 263-282, Robotics Society of Japan, December 1990.

[17] Kwak, S. H. and McGhee, R. B., *Rule-Based Motion Coordination for the Adaptive Suspension Vehicle*, Technical Report, No. NPS52-88-011, Naval Postgraduate School, Monterey, CA, May 1988.

[18] Bromley, H. and Lamson, R. *Lisp Lore*, 2nd edition, Kluwer Academic Publishers, New York, 1987.

[19] Lee, W. J., and Orin, D. E., "Omnidirectional Supervisory Control of a Multilegged Vehicle Using Periodic Gaits," *IEEE Transactions on Robotics and Automation*, Vol. RA-4, No. 2, pp. 635-642, December 1988.

[20] Bawden, A., Greenblatt, J., Holloway, T., Knight, D., Moon, D, and Weinreb, D., "The Lisp Machine," *An MIT Perspective*, pp. 343-373, ed. by P.H. Winston and R.H. Brown, 1979.

[21] Anon., *User's Guide to Symbolics Computers*, Symbolics, Inc., Concord, Massachusetts, July, 1986.

[22] Anon., *User's Guide to Symbolics Prolog*, Symbolics, Inc., Concord, Massachusetts, September, 1986.

[23] Winston, P. H., *Artificial Intelligence*, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1984.

[24] Lee, W. J., and Orin, D. E., "The Kinematics of Motion Planning for Multilegged Vehicles over Uneven Terrain," *IEEE Transactions on Robotics and Automation*, Vol. RA-4, No.2 3, pp. 204-212, April, 1988.

[25] Hirose, S., Fukuda, Y., and Kikuchi, H., "The Gait Control System of a Quadruped Walking Machine," *Advanced Robotics*, Vol. 1, No. 4, December 1986, pp. 289-323.

[26] Klein, C. A. and Messuri, D. A., "Automatic Body Regulation for Maintaining Stability of a Legged Vehicle During Rough-Terrain Locomotion," *IEEE Journal of Robotics and Automation*, Vol. RA-1, No. 3, pp. 132-141, September 1985.

[27] Dobry, T., *The X-1: A High Performance Prolog Machine*, Technical Report, Xenologic Inc., Newark, California, 1989.

[28] Gelsinger, P. P., Gargini, P. A., Parker, G. H., and Yu, A. Y., "Microprocessors Circa 2000," *IEEE Spectrum*, Vol. 26, No. 10, pp. 43-47, October 1989.

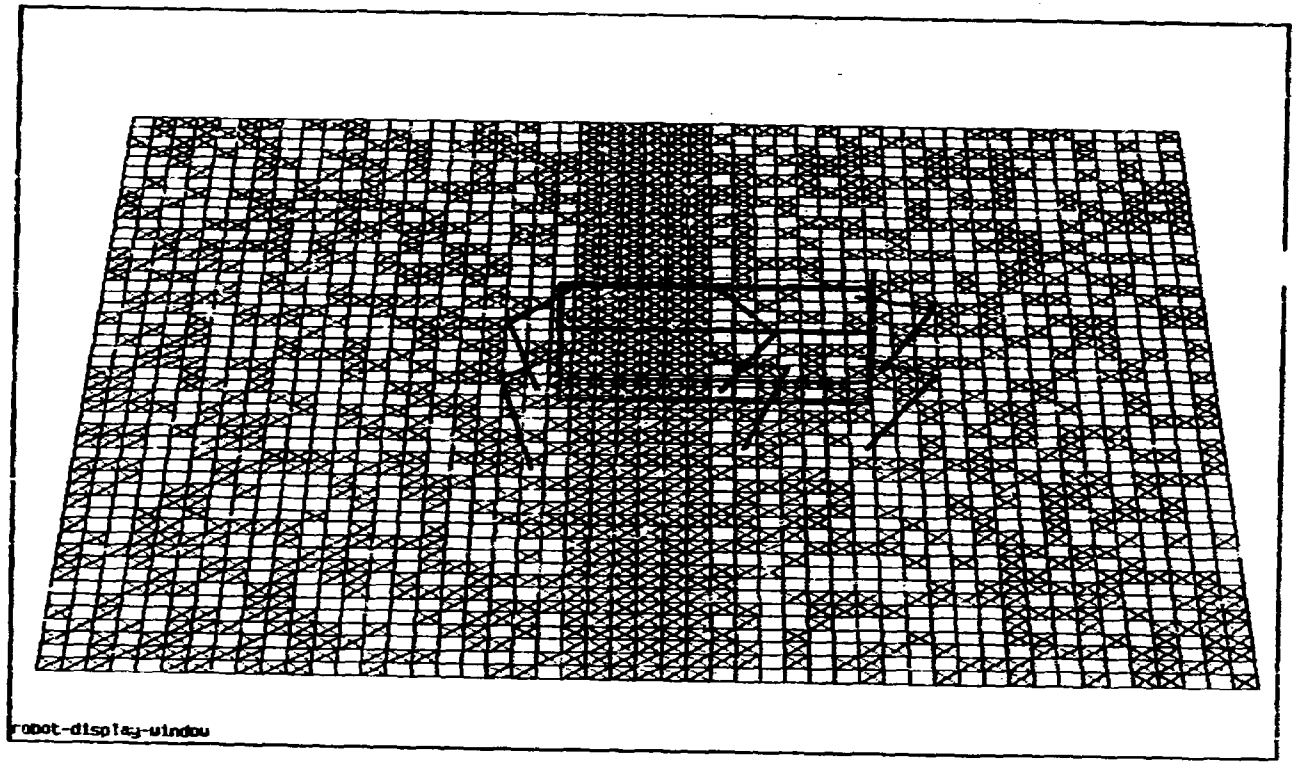


Figure 1: Typical Simulation Terrain and Vehicle

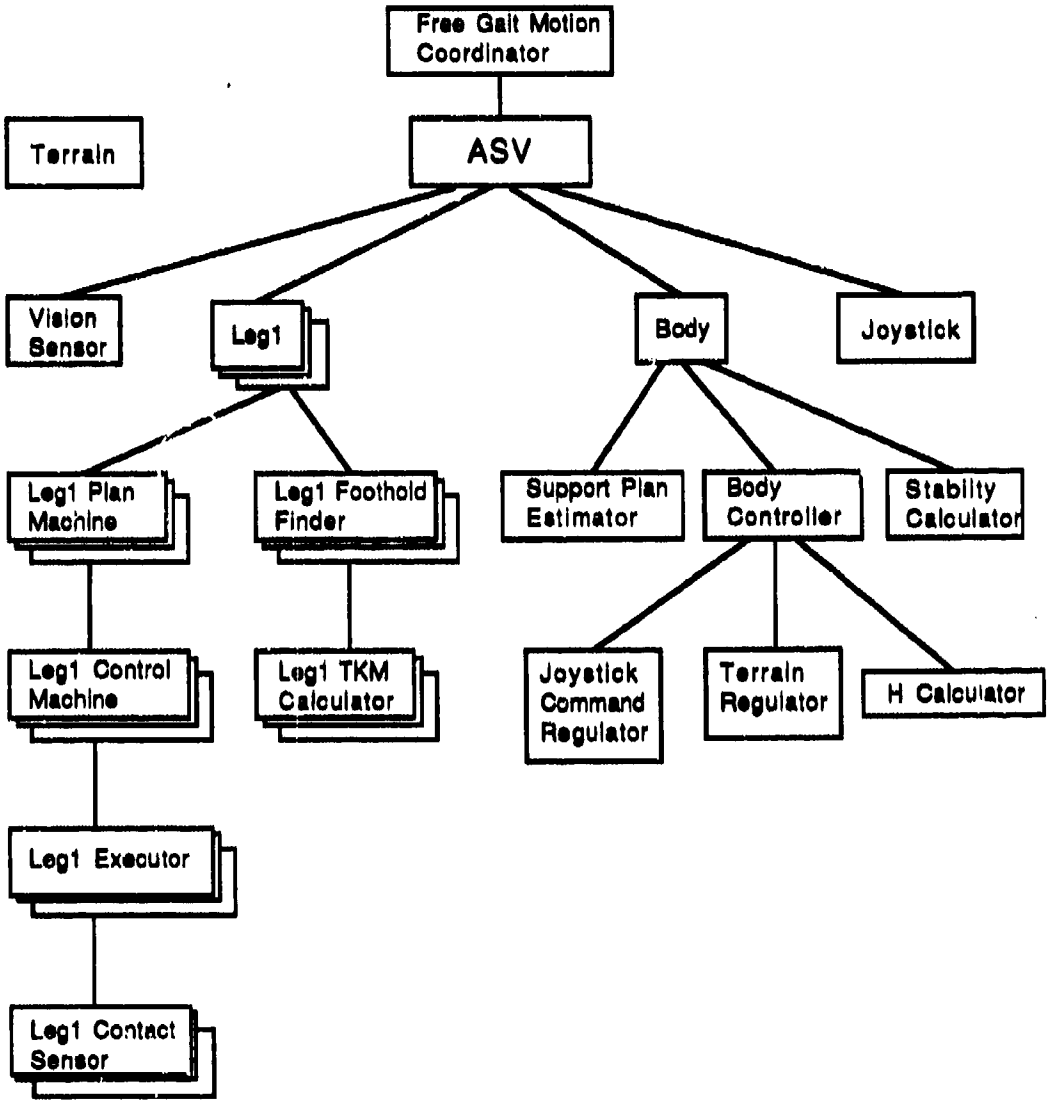


Figure 2: Hierarchy of simulation objects

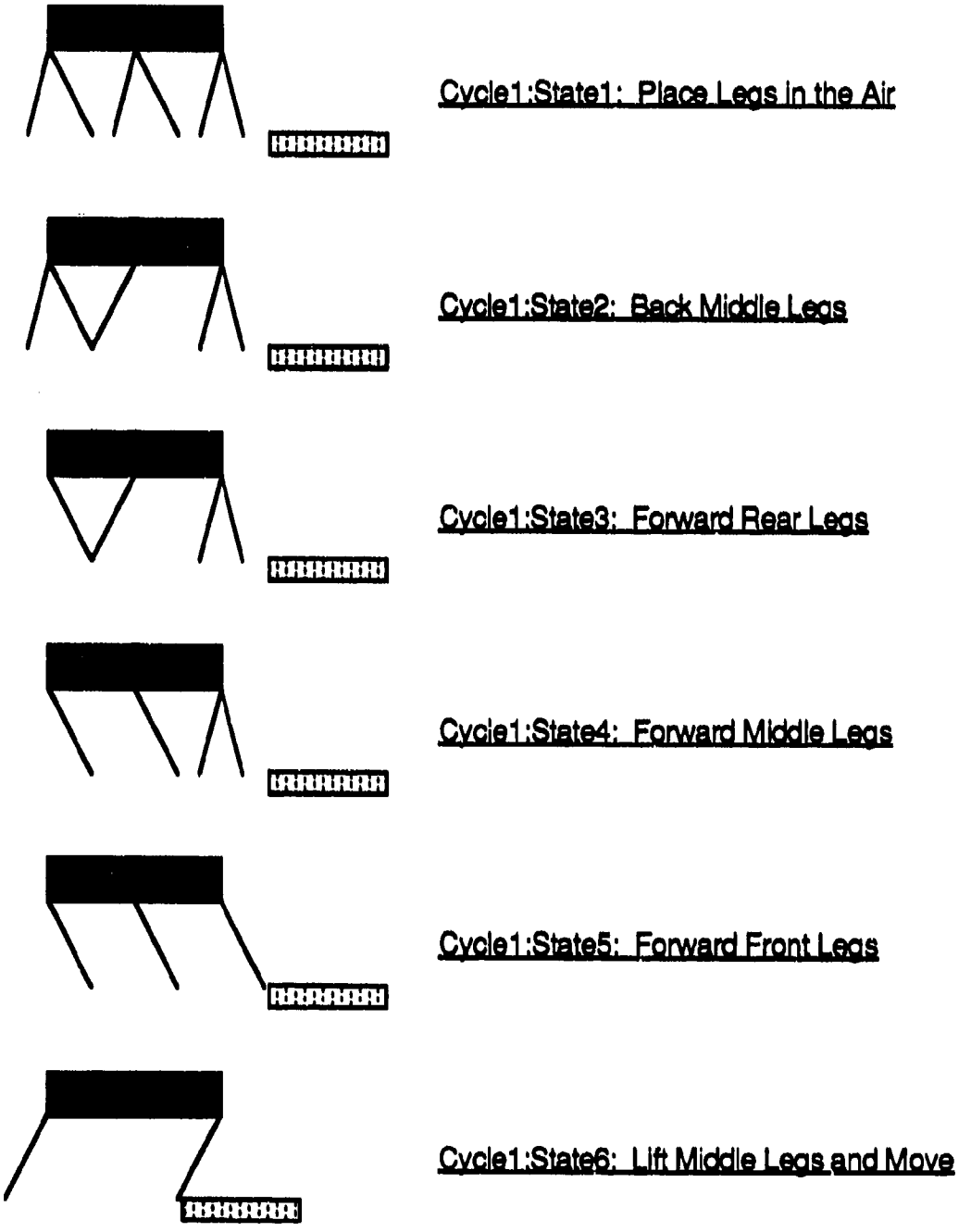
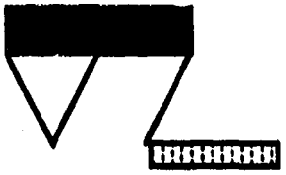
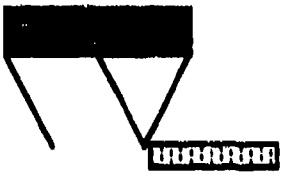


Figure 3: Ditch Crossing Preparation Phase

Cycle2:State1: Back Middle Legs

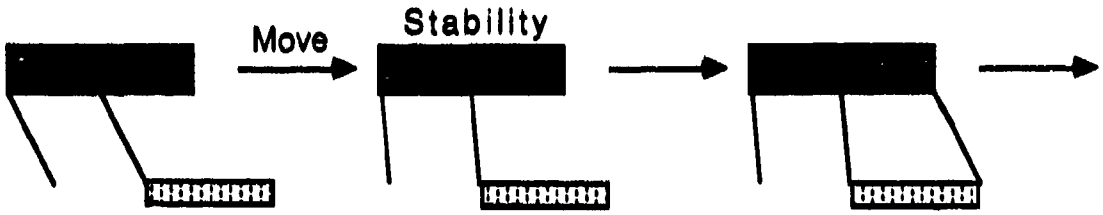


Cycle2:State2: Forward Rear Legs

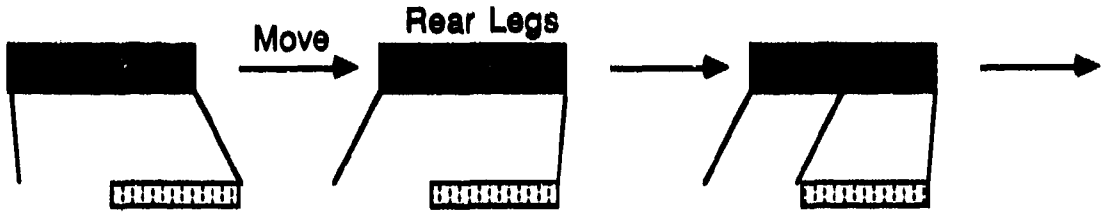


Cycle2:State3: Forward Middle Legs

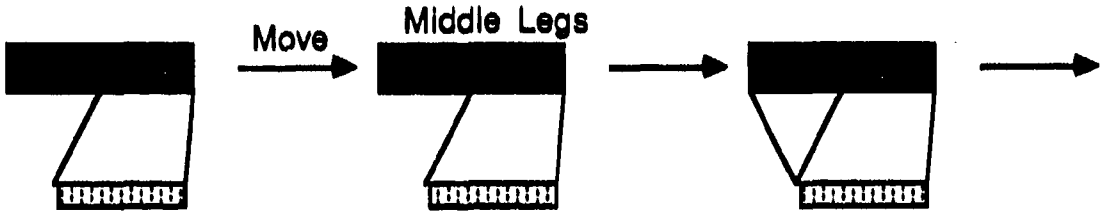
Figure 3: Continued ...



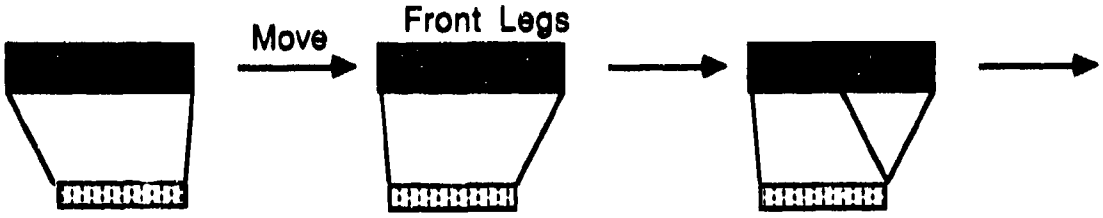
Cycle3:State1: Move Forward Front Legs



Cycle3:State2: Move Back Middle Legs

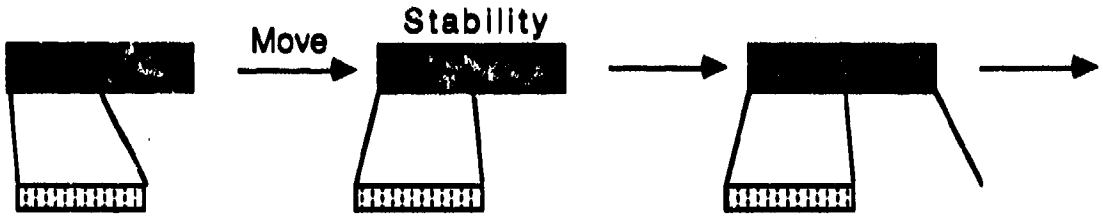


Cycle3:State3: Move Forward Rear Legs

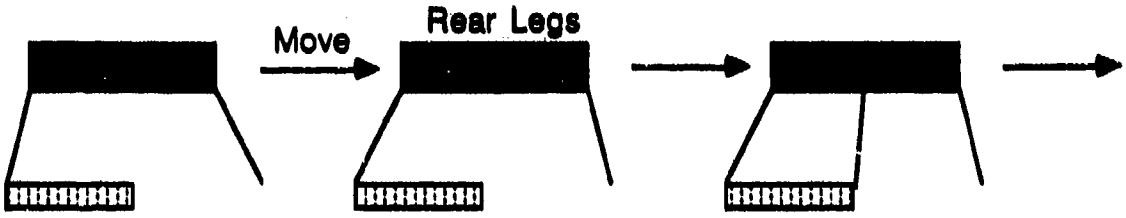


Cycle4:State1: Move Forward Front Legs

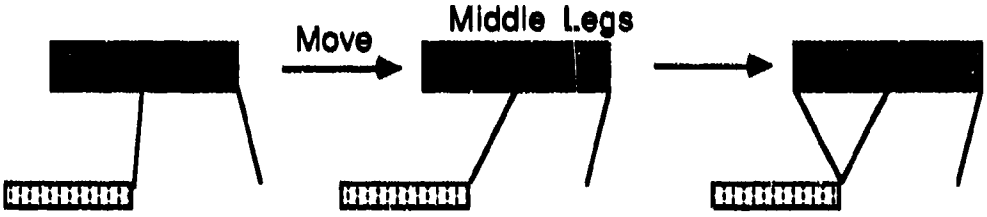
Figure 4: Ditch Crossing Main Phase



Cycle5:State1: Move Forward Front Legs



Cycle5:State2: Move Back Middle Legs



Cycle5:State3: Move Forward Rear Legs

Figure 4: Continued ...

```
;;-*- Mode:PROLOGPackage: robot-rulesBase:10 -*-

robot :- initialize, repeat, my_loop, fail.

initialize :- inits, init_ditch_plan.

init_dicth_plan :- retract(plan_cycle(_)), retract(plan_state(_)), fail.
init_ditch_plan :- asserta(plan_cycle(1)),
                   asserta(plan_state(place_legs_in_the_air)).

my_loop :- get_command, plan, execute, !.

get_command :- X is read_joystick.


plan :- ditch_mode, ditch_crossing_motion_coordination_plan.
plan :- free_gaits_motion_coordination_plan.

ditch_mode :- ditch_mode(in).   ;cleared by ditch_plan.
ditch_mode :- X is at_ditch_area, X == t, asserta(ditch_mode(in)).


execute :- execute_motion, draw_robot, !.

execute_motion :- X is execute_planned_motion.

draw_robot :- X is graphical_display.
```

Figure 5: Prolog Program

```
;;*****************************************************************
;;
;;
;;  Free Gaits Motion Coordination Plan
;;
;;*****************************************************
;;


free_gaits_motion_coordination_plan :- update_robot_state, check_tkm_limit,
                                       leg_plan, body_plan, generate_decision, !.

update_robot_state :- X is update_robot_status.


check_tkm_limit :- A_leg is at_tkm_limit, A_leg \== nil,
                   asserta(limit_leg(A_leg,lift)).
check_tkm_limit.

leg_plan :- lift_a_leg.
leg_plan :- exchange_legs.
leg_plan :- stable.
leg_plan :- place_a_leg.
leg_plan :- wait_for_legs.

stable :- Condition is stable_p, Condition == t.

lift_a_leg :- stable, A_leg is smallest_tkm_leg, A_leg \== nil,
              Condition is stable_without(A_leg), Condition == t,
              asserta(decision(A_leg,_,lift)).

exchange_legs :- stable, LegA is smallest_tkm_leg, LegA \== nil,
                 LegB is max_sm_leg(LegA), LegB \== nil,
                 Condition is has_more_tkm(LegB,LegA),
                 Condition == t,
                 asserta(decision(LegA,LegB,exchange)).

place_a_leg :- A_leg is max_sm_leg(_), A_leg \== nil,
               asserta(decision(A_leg,_,place)).

wait_for_legs :- try_new_foothold.
wait_for_legs :- recovery, asserta(reduce_speed).
wait_for_legs :- asserta(reduce_speed), restore_limit_leg.

try_new_foothold :- A_leg is leg_with_new_foothold, A_leg \== nil,
                    asserta(decision(A_leg,_,place)).

recovery :- A_leg is do_recovery, A_leg \== nil,
            asserta(decision(A_leg,_,place)), restore_limit_leg.

restore_limit_leg :- retract(limit_leg(A_leg,lift)).
restore_limit_leg.
```

Figure 5: Continued ...

```
;;*********************************************************
;;
;;  Ditch Crossing Motion Coordination Plan
;;
;;*********************************************************


ditch_crossing_motion_coordination_plan :- ditch_plan_done, retract(ditch_mode(in)), idle_cycle.
ditch_crossing_motion_coordination_plan :- cycle_planner.



;;
;;********** Cycle Planner ************
;;

ditch_plan_done :- plan_cycle(6), retract(plan_cycle(6)),
                   asserta(plan_cycle(1)),
                   prepare_next_ditch_plan.

prepare_next_ditch_plan :- move.

cycle_planner :- one_cycle_done, plan_cycle(N), N1 is N+1,
                   retract(plan_cycle(N)), asserta(plan_cycle(N1)),
                   idle_cycle.
cycle_planner :- plan_cycle.



;;
;;********** Plan Cycle Dispatcher ************
;;

one_cycle_done :- plan_state(one_plan_cycle_done),
                   retract(plan_state(one_plan_cycle_done)),
                   initialize_plan_state.


initialize_plan_state :- asserta(plan_state(start)).


plan_cycle :- plan_cycle(1), update_robot_state, ditch_plan_cycle_1,
              body_plan, generate_decision,!.
plan_cycle :- plan_cycle(2), update_robot_state, ditch_plan_cycle_2,
              body_plan, generate_decision,!.
plan_cycle :- plan_cycle(3), update_robot_state, ditch_plan_cycle_3,
              body_plan, generate_decision,!.
plan_cycle :- plan_cycle(4), update_robot_state, ditch_plan_cycle_4,
              body_plan, generate_decision,!.
plan_cycle :- plan_cycle(5), update_robot_state, ditch_plan_cycle_5,
              body_plan, generate_decision,!.

idle_cycle :- update_robot_state, body_plan, generate_decision, !.
```

Figure 5: Continued ...

```
;;
;;********** Cycles *********
;;


ditch_plan_cycle_1 :- plan_state(start), retract(plan_state(start)),
                      asserta(plan_state(place_legs_in_the_air)),
                      place_legs_in_the_air(back_middle_legs).
ditch_plan_cycle_1 :- place_legs_in_the_air(back_middle_legs).
ditch_plan_cycle_1 :- back_middle_legs(forward_rear_legs).
ditch_plan_cycle_1 :- forward_rear_legs(forward_middle_legs).
ditch_plan_cycle_1 :- forward_middle_legs(forward_front_legs).
ditch_plan_cycle_1 :- forward_front_legs(lift_middle_legs_and_move).
ditch_plan_cycle_1 :- lift_middle_legs_and_move(one_plan_cycle_done).


ditch_plan_cycle_2 :- plan_state(start), retract(plan_state(start)),
                      asserta(plan_state(back_middle_legs)),
                      back_middle_legs(forward_rear_legs).
ditch_plan_cycle_2 :- back_middle_legs(forward_rear_legs).
ditch_plan_cycle_2 :- forward_rear_legs(forward_middle_legs).
ditch_plan_cycle_2 :- forward_middle_legs(one_plan_cycle_done).


ditch_plan_cycle_3 :- plan_state(start), retract(plan_state(start)),
                      asserta(plan_state(move_forward_front_legs)),
                      move_forward_front_legs(move_forward_middle_legs).
ditch_plan_cycle_3 :- move_forward_front_legs(move_back_middle_legs).
ditch_plan_cycle_3 :- move_back_middle_legs(move_forward_rear_legs).
ditch_plan_cycle_3 :- move_forward_rear_legs(one_plan_cycle_done).


ditch_plan_cycle_4 :- plan_state(start), retract(plan_state(start)),
                      asserta(plan_state(move_forward_middle_legs)),
                      move_forward_middle_legs(one_plan_cycle_done).
ditch_plan_cycle_4 :- move_forward_middle_legs(one_plan_cycle_done).


ditch_plan_cycle_5 :- plan_state(start), retract(plan_state(start)),
                      asserta(plan_state(move_forward_front_legs)),
                      move_forward_front_legs(move_forward_middle_legs).
ditch_plan_cycle_5 :- move_forward_front_legs(move_back_middle_legs).
ditch_plan_cycle_5 :- move_back_middle_legs(move_forward_rear_legs).
ditch_plan_cycle_5 :- move_forward_rear_legs(one_plan_cycle_done).
```

Figure 5: Continued ...

```
;;
;;*************** States ***************
;;


;;;;; back_middle_legs subgroup

back_middle_legs(Next_State) :- plan_state(back_middle_legs),
                                back_middle_legs_done,
                                retract(plan_state(back_middle_legs)),
                                asserta(plan_state(Next_State)),
                                 stop.
back_middle_legs(Next_State) :- plan_state(back_middle_legs),
                                do_back_middle_legs,
                                 stop.


;;;;; forward_front_legs subgroup

forward_front_legs(Next_State) :- plan_state(forward_front_legs)),
                                  forward_front_legs_done,
                                  retract(plan_state(forward_front_legs),
                                  asserta(plan_state(Next_State)),
                                  stop.
forward_front_legs(Next_State) :- plan_state(forward_front_legs)),
                                  do_forward_front_legs,
                                  stop.


;;;;; forward_middle_legs subgroup

forward_middle_legs(Next_State) :- plan_state(forward_middle_legs),
                                  forward_middle_legs_done,
                                  retract(plan_state(forward_middle_legs),
                                  asserta(plan_state(Next_State)),
                                  stop.
forward_middle_legs(Next_State) :- plan_state(forward_middle_legs),
                                  do_forward_middle_legs,
                                  stop.


;;;;; forward_rear_legs subgroup

forward_rear_legs(Next_State) :- plan_state(forward_rear_legs),
                              forward_rear_legs_done,
                              retract(plan_state(forward_rear_legs),
                              asserta(plan_state(Next_State)),
                              stop.
forward_rear_legs(Next_State) :- plan_state(forward_rear_legs),
                              do_forward_rear_legs,
                              stop.


;;;;; lift_middle_legs_and_move subgroup

lift_middle_legs_and_move(Next_State) :- plan_state(lift_middle_legs_and_move),
                                  move_done, stop,
```

Figure 5: Continued ...

```
                                        retract(plan_state(lift_middle_legs_and_move)),
                                        asserta(plan_state(Next_State)).
lift_middle_legs_and_move(Next_State) :- plan_state(lift_middle_legs_and_move),
                                        do_lift_middle_legs, move.


;;;;; move_back_middle_legs subgroup

move_back_middle_legs(Next_State) :- plan_state(move_back_middle_legs),
                                        move_back_middle_legs_done,
                                        retract(plan_state(move_back_middle_legs)),
                                        asserta(plan_state(Next_State)).
move_back_middle_legs(Next_State) :- plan_state(move_back_middle_legs),
                                        do_move_back_middle_legs.


;;;;; move_forward_front_legs subgroup

move_forward_front_legs(Next_State) :- plan_state(move_forward_front_legs),
                                        move_forward_front_legs_done,
                                        retract(plan_state(move_forward_front_legs)),
                                        asserta(plan_state(Next_State)).
move_forward_front_legs(Next_State) :- plan_state(move_forward_front_legs),
                                        do_move_forward_front_legs.


;;;;; move_forward_middle_legs subgroup

move_forward_middle_legs(Next_State) :- plan_state(move_forward_middle_legs),
                                        move_forward_middle_legs_done,
                                        retract(plan_state(move_forward_front_legs)),
                                        asserta(plan_state(Next_State)).
move_forward_middle_legs(Next_State) :- plan_state(move_forward_middle_legs),
                                        do_move_forward_middle_legs.


;;;;; move_forward_rear_legs subgroup

move_forward_rear_legs(Next_State) :- plan_state(move_forward_rear_legs),
                                        move_forward_middle_legs_done,
                                        retract(plan_state(move_forward_rear_legs)),
                                        asserta(plan_state(Next_State)).
move_forward_rear_legs(Next_State) :- plan_state(move_forward_rear_legs),
                                        do_move_forward_rear_legs.


;;;;; place_legs_in_the_air subgroup

place_legs_in_the_air(Next_State) :- plan_state(place_legs_in_the_air),
                                        place_legs_in_the_air_done,
                                        retract(plan_state(place_legs_in_the_air)),
                                        asserta(plan_state(Next_state)),
                                        stop.
place_legs_in_the_air(Next_State) :- plan_state(place_legs_in_the_air),
                                        do_place_legs_in_the_air, stop.
```

Figure 5: Continued ...

```
;;
;;**************** State Executors ******************
;;
;;


;;;;; do_back_middle_legs subgroup

back_middle_legs_done :- all_middle_legs_lifted, all_middle_legs_placed,
                         clear_middle_lifted_memory, clear_move_memory.
do_back_middle_legs :- all_middle_legs_lifted, place_middle_legs_back.
do_back_middle_legs :- lift_middle_legs.

all_middle_legs_lifted :- middle_legs(lifted).
all_middle_legs_lifted :- X is both_middle_legs_lifted, X == t,
                          asserta(middle_legs(lifted)).

all_middle_legs_placed :- X is both_middle_legs_placed, X == t.

clear_middle_lifted_memory :- retract(middle_legs(lifted)).

place_middle_legs_back :- A_leg is placable_middle_leg, A_leg \== nil,
                          asserta(decision(A_leg,_,place_back)).
place_middle_legs_back.

lift_middle_legs :- A_leg is liftable_middle_leg, A_leg \== nil,
                    asserta(decision(A_leg,_,lift)).
lift_middle_legs.


;;;;; do_forward_front_legs subgroup

forward_front_legs_done :- all_front_legs_lifted, all_front_legs_placed,
                           clear_front_lifted_memory, clear_move_memory.

do_forward_front_legs :- all_front_legs_lifted, place_front_legs.
do_forward_front_legs :- lift_front_legs.

all_front_legs_lifted :- front_legs(lifted).
all_front_legs_lifted :- X is both_front_legs_lifted, X == t,
                         asserta(front_legs(lifted)).

all_front_legs_placed :- X is both_front_legs_placed, X == t.

clear_front_lifted_memory :- retract(front_legs(lifted)).

place_front_legs :- A_leg is placable_front_leg, A_leg \== nil,
                    asserta(decision(A_leg,_,place)).
place_front_legs.

lift_front_legs :- A_leg is liftable_front_leg, A_leg \== nil,
                   asserta(decision(A_leg,_,lift)).
lift_front_legs.
```

Figure 5: Continued ...

```
;;;;; do_forward_middle_legs subgroup

forward_middle_legs_done :- all_middle_legs_lifted, all_middle_legs_placed,
                            clear_middle_lifted_memory, clear_move_memory.

do_forward_middle_legs :- all_middle_legs_lifted, place_middle_legs.
do_forward_middle_legs :- lift_middle_legs.

place_middle_legs :- A_leg is placable_middle_leg, A_leg \== nil,
                     asserta(decision(A_leg,_,place)).
place_middle_legs.


;;;;; do_forward_rear_legs subgroup

forward_rear_legs_done :- all_rear_legs_lifted, all_rear_legs_placed,
                          clear_rear_lifted_memory, clear_move_memory.

do_forward_rear_legs :- all_front_legs_lifted, place_rear_legs.
do_forward_rear_legs :- lift_rear_legs.

all_rear_legs_lifted :- rear_legs(lifted).
all_rear_legs_lifted :- X is both_rear_legs_lifted, X == t,
                        asserta(rear_legs(lifted)).

all_rear_legs_placed :- X is both_rear_legs_placed, X == t.

clear_rear_lifted_memory :- retract(rear_legs(lifted)).

place_rear_legs :- A_leg is placable_rear_leg, A_leg \== nil,
                   asserta(decision(A_leg,_,place)).
place_rear_legs.

lift_rear_legs :- A_leg is liftable_rear_leg, A_leg \== nil,
                  asserta(decision(A_leg,_,lift)).
lift_rear_legs.


;;;;; do_lift_middle_legs subgroup

do_lift_middle_legs :- lift_middle_legs.


;;;;; do_move_back_middle_legs subgroup

move_back_middle_legs_done :- all_middle_legs_lifted, all_middle_legs_placed,
                              clear_middle_lifted_memory, clear_move_memory,
                              stop.
do_move_back_middle_legs :- all_middle_legs_lifted, move_done, stop,
                            place_middle_legs_back.
do_move_back_middle_legs :- all_middle_legs_lifted, move.
do_move_back_middle_legs :- lift_middle_legs, stop.
```

Figure 5: Continued ...

```
;;;;; do_move_forward_front_legs subgroup

move_forward_front_legs_done :- all_front_legs_lifted, all_front_legs_placed,
                                clear_front_lifted_memory, clear_move_memory,
                                stop.
do_move_forward_front_legs :- all_front_legs_lifted, move_done, stop,
                              place_front_legs.
do_move_forward_front_legs :- all_front_legs_lifted, move.
do_move_forward_front_legs :- lift_front_legs, stop.


;;;;; do_move_forward_middle_legs subgroup

move_forward_middle_legs_done :- all_middle_legs_lifted,
                                 all_middle_legs_placed,
                                 clear_middle_lifted_memory,
                                 clear_move_memory, stop.
do_move_forward_middle_legs :- all_middle_legs_lifted, move_done, stop,
                               place_middle_legs.
do_move_forward_middle_legs :- all_middle_legs_lifted, move.
do_move_forward_middle_legs :- lift_middle_legs, stop.


;;;;; do_move_forward_rear_legs subgroup

move_forward_rear_legs_done :- all_rear_legs_lifted, all_rear_legs_placed,
                               clear_rear_lifted_memory, clear_move_memory,
                               stop.
do_move_forward_rear_legs :- all_rear_legs_lifted, move_done, stop,
                             place_rear_legs.
do_move_forward_rear_legs :- all_rear_legs_lifted, move.
do_move_forward_rear_legs :- lift_rear_legs, stop.


;;;;; place_legs_in_the_air subgroup

place_legs_in_the_air_done :- X is all_legs_placed, X == t.

place_legs_in_the_air :- A_leg is placable_leg, A_leg \== nil,
                         asserta(decision(A_leg,_,place)).
place_legs_in_the_air.


;;;;; body_movement subgroup

move :- asserta(resume_movement).
stop :- asserta(stop_movement).

clear_move_memory :- retract(move(done)).
clear_move_memory.

move_done :- move(done).
move_done :- X is at_tkm_limit, X \== nil, asserta(move(done)).
move_done :- X is at_stability_limit, X \== nil, asserta(move(done)).
```

Figure 5: Continued ...

```
;;*************************************************************
;;
;;          Plan Libraries
;;
;;*************************************************************

;;;;; body_plan subgroup

body_plan :- speed_plan, trajectory_plan.

speed_plan :- retract(reduce_speed), slow_down.
speed_plan :- speed_up.

speed_up :- X is speed_up_robot.

slow_down :- X is slow_down_robot.

trajectory_plan :- stable_m, restore_trajectory.
trajectory_plan :- modify_trajectory.

stable_m :- Condition is stable_p_m, Condition == t.

restore_trajectory :- X is restore_command.

modify_trajectory :- X is modify_command.


;;;;; generate_decision subgroup

generate_decision :- retract(decision(A_leg,B_leg,A_decision)),
                     X is send_decision(A_leg,B_leg,A_decision), fail.
generate_decision :- retract(limit_leg(A_leg,A_decision)),
                     X is send_decision(A_leg,_,A_decision), fail.
generate_decision.
```

Figure 5: Continued ...

Appendix

Lisp Code for ASV Simulation

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************************
;
;  body-controller definition
;
;*****************************************************************

(defflavor body-controller(joystick-command-regulator terrain-regulator
                           H-calculator
                           body-trans-rate1 body-rotate-rate1
                           body-trans-rate6 body-rotate-rate6
                           body-trans-rate10 body-rotate-rate10
                           H1 inv-H1 H6 inv-H6 H10 inv-H10
                           H inv-H body-t body-r)

           ()
  :initable-instance-variables)



(defmethod (body-controller :initti)
           ()
  (setf joystick-command-regulator (make-instance 'joystick-command-regulator))
  (setf terrain-regulator (make-instance 'terrain-regulator))
  (setf H-calculator (make-instance 'H-calculator))
  (send joystick-command-regulator :initti)
  (send terrain-regulator :initti)
  (setf H (send H-calculator :initti))
  (send self :init-body-rates)
  (send self :init-H)
  H1
  )


(defmethod (body-controller :init-body-rates)
           ()
  (setf body-trans-rate1 '(0 0 0))
  (setf body-trans-rate6 '(0 0 0))
  (setf body-trans-rate10 '(0 0 0))
  (setf body-rotate-rate1 '(0 0 0))
  (setf body-rotate-rate6 '(0 0 0))
  (setf body-rotate-rate10 '(0 0 0))
  )



(defmethod (body-controller :init-H)
           ()
; library fucntion : ident
  (setf H1  H)
  (setf H6  H)
  (setf H10 H)
  (setf inv-H (matrixinv H))
  (setf inv-H1 inv-H)
  (setf inv-H6 inv-H)
  (setf inv-H10 inv-H))
```

```
(defmethod (body-controller :control)
           (joystick-command deceleration-factor estimated-support-plane)
  (setf H H1)
  (send self :update joystick-command deceleration-factor estimated-support-plane)
  (send self :save)
  (dotimes (i 10)
    (cond ((equal i 0)
           (setf body-trans-rate1 body-t)
           (setf body-rotate-rate1 body-r)
           (setf H1 H)
           (setf inv-H1 inv-H))
          ((equal i 5)
           (setf body-trans-rate6 body-t)
           (setf body-rotate-rate6 body-r)
           (setf H6 H)
           (setf inv-H6 inv-H))
          ((equal i 9)
           (setf body-trans-rate10 body-t)
           (setf body-rotate-rate10 body-r)
           (setf H10 H)
           (setf inv-H10 inv-H)))
    (send self :update joystick-command deceleration-factor estimated-support-plane)
    )
  (send self :restore))


(defmethod (body-controller :update)
           (joystick-command deceleration-factor estimated-support-plane)
; internally used by control method
  (let* ((t-command (send terrain-regulator :regulate
                                  estimated-support-plane H))
         (j-command (send joystick-command-regulator :regulate
                                  joystick-command deceleration-factor))
         )
    (setf body-t (list (first j-command) (second j-command)
                       (third t-command)))
    (setf body-r (list (first t-command) (second t-command)
                       (third j-command)))
    (setf H (send H-calculator :new-H body-t body-r))
    (setf inv-H (matrixinv H))))


(defmethod (body-controller :restore)
           ()
; internally used by control method
  (send joystick-command-regulator :restore)
  (send terrain-regulator :restore)
  (send H-calculator :restore))


(defmethod (body-controller :save)
           ()
; internally used by control method
  (send joystick-command-regulator :save)
  (send terrain-regulator :save)
  (send H-calculator :save))
```

```
(defmethod (body-controller :get-body-trans-rate1)
           ()
  body-trans-rate1)

(defmethod (body-controller :get-body-rotate-rate1)
           ()
  body-rotate-rate1)

(defmethod (body-controller :get-body-trans-rate10)
           ()
  body-trans-rate10)

(defmethod (body-controller :get-body-rotate-rate10)
           ()
  body-rotate-rate10)

(defmethod (body-controller :get-H1)
           ()
  H1)

(defmethod (body-controller :get-inv-H1)
           ()
  inv-H1)

(defmethod (body-controller :get-H6)
           ()
  H6)

(defmethod (body-controller :get-inv-H6)
           ()
  inv-H6)

(defmethod (body-controller :get-H10)
           ()
  H10)

(defmethod (body-controller :get-inv-H10)
           ()
  inv-H10)
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*********************************************************
;
;  body flavor definition
;
;*********************************************************



(defflavor body(stability-calculator support-plane-estimator
                                     body-controller owner
                                     estimated-support-plane
                                     deceleration-factor
                                     support-plane-clock
                                     modify-vector
                                     modify-vector-p
                                     stop-motion-flag
                                     joy-command)
           ()
  :initable-instance-variables)


(defmethod (body :slow-down)
           ()
  (setf deceleration-factor (+ deceleration-factor 1))
  (if (> deceleration-factor 20)
      (setf deceleration-factor 20)))

(defmethod (body :speed-up)
           ()
  (setf deceleration-factor (- deceleration-factor 1))
  (if (< deceleration-factor 0)
      (setf deceleration-factor 0)))


(defmethod (body :stable-m)
           (supporting-legs)
  (send stability-calculator :stable-m
            supporting-legs (send body-controller :get-H10)))


(defmethod (body :stable-p-m)
           (supporting-p-legs a-leg)
  (send stability-calculator :stable-p-m
                       supporting-p-legs
                       (send body-controller :get-H1)))


(defmethod (body :stop-p)
           ()
  (let ((trans-rate (send self :get-body-trans-rate1)))
    (equal (list (first trans-rate)
                 (second trans-rate))
           '(0.0 0.0))))


(defmethod (body :modify-command)
           ()
  (setf modify-vector
```

```
        (send stability-calculator :get-recovery-vector)))

(defmethod (body :modify-command-p)
           ()
  (setf modify-vector-p
        (send stability-calculator :get-recovery-vector-p)))

(defmethod (body :restore-command)
           ()
  (setf modify-vector '(0 0 0)))


(defmethod (body :restore-command-p)
           ()
  (setf modify-vector-p '(0 0 0)))

(defmethod (body :stop-motion)
           (a-leg)
  (setf stop-motion-flag a-leg))


(defmethod (body :restore-motion)
           ()
  (setf stop-motion-flag nil))



(defmethod (body :initti)
           ()
  (setf deceleration-factor 0)
  (setf modify-vector-p '(0 0 0))
  (setf modify-vector '(0 0 0))
  (setf stop-motion-flag nil)
  (setf support-plane-clock 10)
  (setf stability-calculator
        (make-instance 'stability-calculator))
  (setf support-plane-estimator
        (make-instance 'support-plane-estimator))
  (setf body-controller
        (make-instance 'body-controller))
  (send stability-calculator :initti)
  (send support-plane-estimator :initti)
  (send body-controller :initti)
  )

(defmethod (body :get-modify-vector)
           ()
  (vectsub modify-vector
          (dotprod modify-vector
                   (normalize-vector joy-command))))


(defmethod (body :get-modify-vector-p)
           ()
  modify-vector-p)
```

```
(defmethod (body :calculate-motion)
           (joystick-command legs)
  (setf joy-command joystick-command)
  (cond ((equal support-plane-clock 10)
; ??? bug ???
         (setf estimated-support-plane
               (send support-plane-estimator :get-plane legs))
         (setf support-plane-clock 0)))
  (setf support-plane-clock (+ support-plane-clock 1))
  (cond
    ((or stop-motion-flag (null modify-vector-p))
     (send body-controller :control
           '(0 0 0)
           0 estimated-support-plane))
    (modify-vector-p
     (send body-controller :control
           (vectadd joy-command (send self :get-modify-vector-p))
           deceleration-factor estimated-support-plane))
    (t
     (control body-controller
           (vectadd joy-command (send self :get-modify-vector))
           deceleration-factor estimated-support-plane))))


(defmethod (body :get-estimated-support-plane)
           ()
  estimated-support-plane)


(defmethod (body :get-body-trans-rate1)
           ()
  (send body-controller :get-body-trans-rate1))


(defmethod (body :get-body-rotate-rate1)
           ()
  (send body-controller :get-body-rotate-rate1))


(defmethod (body :get-body-trans-rate10)
           ()
  (send body-controller :get-body-trans-rate10))


(defmethod (body :get-body-rotate-rate10)
           ()
  (send body-controller :get-body-rotate-rate10))


(defmethod (body :get-H1)
           ()
  (send body-controller :get-H1))


(defmethod (body :get-inv-H1)
```

```
            ()
  (send body-controller :get-inv-H1))


(defmethod (body :get-H6)
            ()
  (send body-controller :get-H6))


(defmethod (body :get-inv-H6)
            ()
  (send body-controller :get-inv-H6))


(defmethod (body :get-H10)
            ()
  (send body-controller :get-H10))


(defmethod (body :get-inv-H10)
            ()
  (send body-controller :get-inv-H10))





(defmethod (body :more-stable)
            (supporting-legs leg1 leg2)
  (send stability-calculator :more-stable
                supporting-legs (send body-controller :get-H10)
                leg1 leg2))


(defmethod (body :stable)
            (supporting-legs)
  (send stability-calculator :stable
          supporting-legs (send body-controller :get-H10)))


(defmethod (body :stable-p)
            (supporting-p-legs)
  (send stability-calculator :stable-p
          supporting-p-legs (send body-controller :get-H1)))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*********************************************************
;
;  regulator flavor definition
;
;*********************************************************


(defflavor regulator((max-a 3.2174) (time-const 0.5) (sampling-time 0.1))
           ()
  :initable-instance-variables)


(defmethod (regulator :filter)
           (desired-x present-x)
; first order regulation
  (let ((del-vel (/ (- desired-x present-x) time-const)))
    (+ (* (send self :g-limitor del-vel) sampling-time)
       present-x)))


(defmethod (regulator :g-limitor)
           (del-vel)
; limit acceleration to 3.2174 ft/(sec*sec)  or 0.1 G.
  (cond ((> del-vel max-a)   max-a)
        ((< del-vel (- max-a)) (- max-a))
        (T del-vel)))




;*********************************************************
;
;  joystick-command-regulator flavor definition
;
;*********************************************************


(defflavor joystick-command-regulator(body-trans-rate-x
                                      body-trans-rate-y
                                      body-rotate-rate-z
                                      old-body-trans-rate-x
                                      old-body-trans-rate-y
                                      old-body-rotate-rate-z)
           (regulator)
  :initable-instance-variables)


(defmethod (joystick-command-regulator :initti)
           ()
  (setf body-trans-rate-x 0.0)
  (setf body-trans-rate-y 0.0)
  (setf body-rotate-rate-z 0.0)
  (list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))
```

```
(defmethod (joystick-command-regulator :regulate)
           (joystick-command deceleration-factor)
  (if (<= deceleration-factor 0)
      (setf deceleration-factor 0.5))  ; remove effect of deceleration-factor.
  (let* ((d-const 0.5)
         (x (* (first  joystick-command) (/ d-const deceleration-factor)))
         (y (* (second joystick-command) (/ d-const deceleration-factor)))
         (r (* (third  joystick-command) (/ d-const deceleration-factor))))
    (setf body-trans-rate-x  (send self :filter x body-trans-rate-x))
    (setf body-trans-rate-y  (send self :filter y body-trans-rate-y))
    (setf body-rotate-rate-z (send self :filter r body-rotate-rate-z)))
    (if (< (abs body-trans-rate-x)  0.02)  (setf body-trans-rate-x  0.0))
    (if (< (abs body-trans-rate-y)  0.02)  (setf body-trans-rate-y  0.0))
    (if (< (abs body-rotate-rate-z) 0.005)  (setf body-rotate-rate-z  0.0))
    (list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))


(defmethod (joystick-command-regulator :restore)
           ()
  (setf body-trans-rate-x old-body-trans-rate-x)
  (setf body-trans-rate-y old-body-trans-rate-y)
  (setf body-rotate-rate-z old-body-rotate-rate-z)
  (list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))


(defmethod (joystick-command-regulator :save)
           ()
  (setf old-body-trans-rate-x body-trans-rate-x)
  (setf old-body-trans-rate-y body-trans-rate-y)
  (setf old-body-rotate-rate-z body-rotate-rate-z)
  (list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*************************************************************
;
;  state flavor definition
;
;*************************************************************

(defflavor state(name next-state)
            ()
  :initable-instance-variables)


(defmethod (state :state-name)
           ()
  name)


(defmethod (state :set-next-state)
           (a-state)
  (setf next-state a-state))




;*************************************************************
;
;  sync-state flavor definition
;
;*************************************************************

(defflavor sync-state((time 0) (del-t 0.1) time-out)
           (state)
  :initable-instance-variables)


(defmethod (sync-state :change)
           ()
  (setf time (+ time del-t))
  (cond ((>= time time-out)
         (setf time 0)
         next-state)
        (t self)))


(defmethod (sync-state :get-time)
           ()
  time)



;*************************************************************
```

```
;
;  async-state flavor definition
;
;*********************************************************


(defflavor async-state((command nil) (observation nil))
           (state)
  :initable-instance-variables)


(defmethod (async-state :change)
           (given-command observed-event)
  (cond ((and (not observation)
              (equal given-command command))
         next-state)
        ((and (not command)
              (equal observed-event observation))
         next-state)
        (t self)))




;*********************************************************
;
;  state-machine flavor definition
;
;*********************************************************


(defflavor state-machine(state owner)
           ()
  :initable-instance-variables)


(defmethod (state-machine :state-name)
           ()
  (send state :state-name))




;*********************************************************
;
;  control-state-machine flavor definition
;
;*********************************************************

(defflavor control-state-machine((command nil) (observation nil)
                                 contact-sensor executor)
           (state-machine)
  :initable-instance-variables)
```

```
(defmethod (control-state-machine :initti)
           (leg-name)
  (if (member leg-name '(leg1 leg4 leg5))
      (send self :init-control-machine 'support)
      (send self :init-control-machine 'ready))
  (setf contact-sensor (send owner :contact-sensor))
  (setf executor (send owner :executor)))


(defmethod (control-state-machine :init-control-machine)
           (a-state-name)
; internally used by init method
  (let (return lift support contact descent advance ready)
    (setf return
          (make-instance 'sync-state
                         :name 'return :time-out 0.6))
    (setf lift
          (make-instance 'sync-state
                         :name 'lift :time-out 0.4
                         :next-state return))
    (setf support
          (make-instance 'async-state
                         :name 'support :command 'recover-command
                         :next-state lift))
    (setf contact
          (make-instance 'sync-state
                         :name 'contact :time-out 1.0
                         :next-state support))
    (setf descent
          (make-instance 'async-state
                         :name 'descent :observation 'contact-confirm
                         :next-state contact))
    (setf advance
          (make-instance 'sync-state
                         :name 'advance :time-out 0.6
                         :next-state descent))
    (setf ready
          (make-instance 'async-state
                         :name 'ready :command 'deploy-command
                         :next-state advance))

    (send return :set-next-state ready)


    (setf state (cond ((equal a-state-name (send ready :state-name))
                       ready)
                      ((equal a-state-name (send advance :state-name))
                       advance)
                      ((equal a-state-name (send descent :state-name))
                       descent)
                      ((equal a-state-name (send contact :state-name))
                       contact)
                      ((equal a-state-name (send support :state-name))
                       support)
                      ((equal a-state-name (send lift :state-name))
                       lift)
                      ((equal a-state-name (send return :state-name))
                       return)))
    )
  )
```

```
;(defmethod (control-state-machine :change :before)
;           ()
;  (cond ((typep state 'async-state)
;         (if (contact-sensor :sensing)
;             (setf observation 'contact-confirm)
;             (setf observation nil))
;          )))


(defmethod (control-state-machine :change)
           ()
    (cond ((typep state 'async-state)
         (if (send contact-sensor :sensing)
             (setf observation 'contact-confirm)
             (setf observation nil))
          ))

  (cond ((typep state 'sync-state)
         (setf state (send state :change)))
        (t (setf state (send state :change command observation))))
;  )


;(defmethod (control-state-machine :change :after)
;           ()
  ; send command to executor with sync-state-time
  (send executor :send-command (send state :state-name))
  (if (typep state 'sync-state)
      (send executor :set-time (send state :get-time))
      (send executor :set-time nil)))



(defmethod (control-state-machine :send-command)
           (a-command)
  (setf command a-command))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*********************************************************

;  display.globals

;*********************************************************

(defvar eye-space nil)
(defvar middle-of-screen nil)


(defvar terrain-joystick)
(defvar graph-terrain)
(defvar graph-asv)


;*********************************************************

;  display.library

;*********************************************************


(defun draw-to-earth (a-point)
  (let ((draw-pt (make-displayable
           middle-of-screen
           (transform eye-space a-point))))
    (draw-to
      (list (truncate (first draw-pt))
        (truncate (second draw-pt)))
      *robot-window*)))


(defun draw-to-earth-d (a-point)
  (let ((draw-pt (make-displayable
           middle-of-screen
           (transform eye-space a-point))))
    (draw-to-d
      (list (truncate (first draw-pt))
        (truncate (second draw-pt)))
      *robot-window*)))


(defun erase-to-earth (a-point)
  (let ((draw-pt (make-displayable
           middle-of-screen
           (transform eye-space a-point))))
    (erase-to
      (list (truncate (first draw-pt))
        (truncate (second draw-pt)))
      *robot-window*)))


(defun eye-trans (eye-pt)
; eye-pt (radius alpha beta)
```

```
; eye:= orient*trans(0,0,-r)*rot(x,-beta)*rot(y,-alpha)*trans(-x,-y,-z)
; returns eye-space
; library : ident, transmat,rotate,matrixmult
  (let* ((orient (ident))
     (rot nil) (trans nil) (eye nil)
     (radius (first eye-pt)) (alpha (second eye-pt)) (beta (third eye-pt))
     (center-of-interest (list (/ (send graph-terrain :max-x) 2)
                               (/ (send graph-terrain :max-y) 2) 0)))
    (setf (aref orient 2 2) -1.0)
    (setf trans (transmat 0 0 (- radius)))
    (setf eye (matrixmult orient trans))
    (setf rot (rotatemat 'y-axis (- alpha)))
    (setf eye (matrixmult eye rot))
    (setf rot (rotatemat 'x-axis (- beta)))
    (setf eye (matrixmult eye rot))
    (setf trans (transmat (- (first center-of-interest))
              (- (second center-of-interest))
              (- (third center-of-interest))))
    (matrixmult eye trans)))


(defun make-displayable  (middle pt)
  (let ((scale   5000.0)
    (x (first pt)) (y (second pt)) (z (third pt)))
    (list (+ (* scale (/ x z)) (first middle))
      (+ (* scale (/ y z)) (second middle)))))


(defun move-to-earth (a-point)
  (let ((draw-pt (make-displayable
           middle-of-screen
           (transform eye-space a-point))))
    (move-to
      (list (truncate (first draw-pt))
        (truncate (second draw-pt))))))





;*************************************************
;
;  joystick flavor definition
;
;*************************************************


(defflavor joystick((joy-x 0) (joy-y 0) (joy-r 0))
           ()
  :initable-instance-variables)


(defmethod (joystick :get-joy-value)
           ()
  (let* ((key-value)
         (delta-x 0.2) (delta-y 0.1) (delta-r 0.01))
    (setf key-value (my-read-char-no-hang))
    (cond ((equal key-value '#\f) (setf joy-x (+ joy-x delta-x)))
          ((equal key-value '#\b)  (setf joy-x (- joy-x delta-x)))
```

```
          ((equal key-value '#\r) (setf joy-y (- joy-y delta-y)))
          ((equal key-value '#\l) (setf joy-y (+ joy-y delta-y)))
          ((equal key-value '#\=)  (setf joy-r (- joy-r delta-r)))
          ((equal key-value '#\-)  (setf joy-r (+ joy-r delta-r))))
    (cond ((>= joy-x 2)  (setf joy-x 2))
          ((<= joy-x -2) (setf joy-x -2)))
    (cond ((>= joy-y 1)  (setf joy-y 1))
          ((<= joy-y -1) (setf joy-y -1)))
    (cond ((>= joy-r 0.1) (setf joy-r 0.1))
          ((<= joy-r -0.1) (setf joy-r -0.1)))
    (cond ((equal key-value '#\q) (setf joy-x 0)
           (setf joy-y 0) (setf joy-r 0)))
    (list joy-x joy-y joy-r (equal key-value '#\x))))


(defmethod (joystick :reset)
        ()
  (setf joy-x 0)
  (setf joy-y 0)
  (setf joy-r 0))


(setf terrain-joystick (make-instance 'joystick))


;*****************************************************
;
;  terrain flavor definition
;
;*****************************************************


(defflavor terrain((terrain-data (make-array '(49 49) :initial-element 0))
                   terrain-height-array terrain-height-list joystick
                   (cursor-x) (cursor-y) (max-x) (max-y)
                   (radius 500) (alpha 0) (beta 0))
           ()
  :initable-instance-variables
  :gettable-instance-variables)


(defmethod (terrain :create)
           ()
  (send self :initti)
  (send self :modify)
  (my-print "Now use joystick to control the robot."))


(defmethod (terrain :kill)
           ()
  (kill-robot-terrain-windows)
  (restore-lisp-listener))


(defmethod (terrain :initti)
           ()
; globals : middle-of-screen, eye-space
  (move-and-shape-lisp-listener)
```

```
  (let ((array-dims (array-dimensions terrain-data)))
    (setf radius 500 alpha 0 beta 0)
    (setf max-x (first array-dims))
    (setf max-y (second array-dims))
    (setf cursor-x (floor (/ max-x 2)))
    (setf cursor-y (floor (/ max-y 2))))
  (setf terrain-height-array (make-array (+ max-x 1)))
  (make-robot-window)
  (setf middle-of-screen
        (middle-of-robot-window))
  (setf eye-space (eye-trans (list 500 0 0)))
  (send self :input-terrain-parameters)
  (my-print "Please use joystick to transform the terrain.")
  (my-print "Wait.")
  (make-visible)
  (send self :erase-obstacles)
  (my-print "Now you can translate the terrain."))



(defmethod (terrain :modify)
           ()
; external : eye-space
  (do ((delta 0.0001)
       (joystick-value nil)
       (end-flag nil))
      (end-flag (my-print "Wait.")
                (send joystick :reset)
                (send self :save-terrain eye-space)
                (send self :draw-obstacles))
    (make-visible)
    (setf joystick-value (send joystick :get-joy-value))
    (let ((x (first joystick-value))
          (y (second joystick-value))
          (r (third joystick-value))
          (fire (fourth joystick-value)))
      (send self :erase-terrain)
      (cond
        (fire (cond ((user-ok)
                     (cond ((user-save)
                            (send self :save-terrain-to-disk (user-file-name))))
                     (setf end-flag t))
                    (t
                     (send joystick :reset)
                     (setf joystick-value (send joystick :get-joy-value)))))
        ((> x delta)      (setf alpha (+ alpha 0.05)))
        ((< x (- delta)) (setf alpha (- alpha 0.05)))
        ((> y delta)      (setf beta  (+ beta  0.05)))
        ((< y (- delta)) (setf beta  (- beta  0.05)))
        ((> r delta)      (setf radius (+ radius 10)))
        ((< r (- delta))  (setf radius (- radius 10))))
      (setf eye-space (eye-trans (list radius alpha beta)))
      (send self :draw-terrain1 eye-space))))


(defmethod (terrain :in-side-of-whole-terrain)
           (a-pos)
  (let ((dimension-terrain (array-dimensions terrain-data))
        (i-x (floor (first a-pos)))
        (i-y (floor (second a-pos))))
    (cond ((< i-x 0) nil)
```

```
          ((< i-y 0) nil)
          ((> i-x (- (first  dimension-terrain) 1)) nil)
          ((> i-y (- (second dimension-terrain) 1)) nil)
          (T))
    ))


(defmethod (terrain :permitted-cell)
           (terrain-pos)
  (let ((i-x (floor (first terrain-pos))) ; find terrain index
        (i-y (floor (second terrain-pos))))
    (if (send self :in-side-of-whole-terrain terrain-pos)
        (if (equal (aref terrain-data i-x i-y) 0) ; permitted
            t
            nil))))


(defmethod (terrain :terrain-point)
           (a-pos-wrt-earth)
  (let* ((x (first a-pos-wrt-earth))
         (y (second a-pos-wrt-earth))
         (z (send self :get-height (list x y))))
    (list x y z)))


(defmethod (terrain :get-height)
           (a-pos-wrt-earth)
; range 0 =< x <= (first dimension-terrain-height), (0 < x <39)
;       0 =< y <= (second dimension-terrain).
  (let* ((dimension-terrain-height (array-dimensions terrain-height-array))
         (x-min 0) (x-max (first dimension-terrain-height))
         (x (first a-pos-wrt-earth)))
    (if (or (< x x-min) (> x x-max))
        -1000
        (let* ((i-x (floor x))              ; get terrain x-index
               (x1 (if (< (- x i-x) 0.5) (- i-x 1) i-x))
               (x2 (if (< (- x i-x) 0.5) i-x (+ i-x 1)))
               (x1 (if (< x1 x-min) 0 x1))
               (x2 (if (>= x2 x-max) (- x-max 1) x2))
               (z1 (aref terrain-height-array x1))
               (z2 (aref terrain-height-array x2))
               (slope (- z2 z1))
               (del-x (- x x1)))
          (+ z1 (* slope del-x))))))


(setf graph-terrain (make-instance 'terrain
                                   :joystick terrain-joystick))


;*****************************************************************

;  terrain.input-terrain-parameters
```

```
;*****************************************************


(defmethod (terrain :input-terrain-parameters)
           ()
  (initialize-menu-variables)
  (cond ((setf *old-terrain-file-name* (get-old-terrain-file-name))
         (send graph-terrain :read-terrain-from-disk *old-terrain-file-name*))
        (t
         (send self :get-new-terrain))))


(defmethod (terrain :get-new-terrain)
           ()
  (send self :get-new-terrain-height)
  (send self :draw-terrain eye-space)
  (send self :set-new-terrain-obstacles)
  (send self :set-new-ditch))


(defmethod (terrain :get-new-terrain-height)
           ()
  (let ((slope-type (get-terrain-slope-type))
        (angle nil) (data nil))
    (cond ((equal slope-type 'single-angle)
           (setf angle (get-terrain-slope-angle)))
          ((equal slope-type 'manual)
           (setf data (get-terrain-slope-data))))
    (my-print "Wait.")
    (send self :read-terrain-height slope-type angle data)))


(defmethod (terrain :set-new-terrain-obstacles)
           ()
  (let ((terrain-type (get-terrain-obstacle-type))
        (values nil)
        (obstacle-ratio nil) (random-seed nil))
    (cond ((equal terrain-type 'random)
           (setf values (get-terrain-random-data))
           (setf obstacle-ratio (first values))
           (setf random-seed (second values))
           (my-print "Wait.")
           (send self :random-terrain obstacle-ratio random-seed))
          (t
           (send self :display-cursor)))))


(defmethod (terrain :set-new-ditch)
           ()
  (let ((ditch-type (get-ditch-type))
        (width-location nil))
    (cond ((equal ditch-type 'add-ditch)
           (setf width-location (get-ditch-width-location))
           (my-print "Wait.")
           (send self :add-ditch (first width-location)
                 (second width-location))
           (send self :draw-obstacles)
           (make-visible))
```

```
          (t nil))))


;*********************************************************

;  terrain.display-terrain

;*********************************************************


(defmethod (terrain :display-cursor)
           ()
  (send self :make-all-permitted)
  (do ((joy-data nil) (x nil) (y nil) (r nil) (fire nil)
       (exit-flag nil))
      (exit-flag (send self :erase-cursor (list cursor-x cursor-y)))
    (make-visible)
    (setf joy-data (send joystick :get-joy-value))
    (setf x (- (second joy-data))) (setf y (first joy-data))
    (setf r (third joy-data)) (setf fire (fourth joy-data))
    (send self :erase-cursor  (list cursor-x cursor-y))
    (cond
      (fire (setf exit-flag t))
      ((> x 0) (setf cursor-x (+ cursor-x 1)) (if (> cursor-x max-x)
                                                  (setf cursor-x max-x)))
      ((< x 0) (setf cursor-x (- cursor-x 1)) (if (< cursor-x 0)
                                                  (setf cursor-x 0)))
      ((> y 0) (setf cursor-y (+ cursor-y 1)) (if (> cursor-y max-y)
                                                  (setf cursor-y max-y)))
      ((< y 0) (setf cursor-y (- cursor-y 1)) (if (< cursor-y 0)
                                                  (setf cursor-y 0)))
      ((< r 0) (setf (aref terrain-data cursor-x cursor-y) 1))
      ((> r 0) (setf (aref terrain-data cursor-x cursor-y) 1)))
    (send self :draw-cursor (list cursor-x cursor-y))
    (send self :draw-obstacles)
    (send joystick :reset)))


(defmethod (terrain :draw-terrain)
           (eye-space)
; external function: \display.library\move-to-earth, draw-to-earth
  (dotimes (x (+ max-x 1))
    (move-to-earth (list x 0 (aref terrain-height-array x)))
    (draw-to-earth (list x max-x (aref terrain-height-array x))))
  (dotimes (y (+ max-y 1))
    (move-to-earth (list 0 y 0))
    (dotimes (x (+ max-x 1))
      (draw-to-earth (list x y (aref terrain-height-array x))))))


(defmethod  (terrain :draw-terrain1)
            (eye-space)
; external function: \display.library\move-to-earth, draw-to-earth
  (do ((xs (list 0 max-x) (cdr xs))
       (x nil))
      ((null xs))
    (setf x (car xs))
    (move-to-earth (list x 0 (aref terrain-height-array x)))
    (draw-to-earth (list x max-x (aref terrain-height-array x))))
```

```
  (do ((ys (list 0 max-y) (cdr ys))
       (y nil))
      ((null ys))
    (setf y (car ys))
    (move-to-earth (list 0 y 0))
    (dotimes (x (+ max-x 1))
      (draw-to-earth (list x y (aref terrain-height-array x))))))


(defmethod (terrain :erase-obstacles)
           ()
; externals : terrain
; external function: \display.library\move-to-earth, draw-to-earth
  (dotimes (i (first (array-dimensions terrain-data)))
    (dotimes (j (second (array-dimensions terrain-data)))
      (cond ((equal 1 (aref terrain-data i j))
         (move-to-earth (list i j))
         (erase-to-earth (list (+ i 1) (+ j 1)))
         (move-to-earth (list (+ i 1) j))
         (erase-to-earth (list i (+ j 1))))))))



(defmethod (terrain :erase-terrain)
           ()
  (clear-robot-window))



(defmethod (terrain :make-all-permitted)
           ()
  (dotimes (i max-x)
    (dotimes (j max-y)
      (setf (aref terrain-data i j) 0))))



(defmethod (terrain :read-terrain-height)
           (terrain-slope-type terrain-slope-angle terrain-slope-data)
    (cond ((equal terrain-slope-type 'default)
           (setf terrain-height-list '((19 0) (25 1) (35 1.5))))
          ((equal terrain-slope-type 'single-angle)
           (let* ((angle (* pi (/ terrain-slope-angle 180)))
                  (max (* 20 (tan angle))))
             (setf terrain-height-list
                   (list '(20 0)
                         (list 40 max)
                         ))))
          (t (setf terrain-height-list terrain-slope-data)))
    (let* ((x1 0) (z1 0) (a-pair) (zz 0)
           (x2 (first (car terrain-height-list)))
           (z2 (second (car terrain-height-list)))
           (slope (/ (- z2 z1) (- x2 x1))))
      (setf terrain-height-list (cdr terrain-height-list))
      (dotimes (i (+ max-x 1))
        (setf zz (+ (* slope (- i x1)) z1))
        (cond ((equal x2 i)
               (setf x1 x2)
               (cond ((setf a-pair (car terrain-height-list))
                      (setf terrain-height-list (cdr terrain-height-list))
                      (setf x2 (first a-pair))
                      (setf z2 (second a-pair))
                      (setf z1 zz)
```

```
                        (setf slope (/ (- z2 z1) (- x2 x1))))
                       (T (setf slope 0) (setf z1 zz)))))
        (setf (aref terrain-height-array i) zz))))



(defmethod (terrain :save-terrain)
           (eye-space)
  (send self :draw-obstacles)
  (send self :draw-terrain eye-space)
  (save-terrain-to-terrain-buffer))


(defmethod (terrain :save-terrain-to-disk)
           (file-name)
  (with-open-file
    (out-file
      (merge-pathnames file-name "robot:kwak.robot.terrain-data  data") :direction :outp
)
    (setf *print-array* t)
    (print terrain-data out-file)
    (print terrain-height-array out-file)
    (print radius out-file)
    (print alpha out-file)
    (print beta out-file)
    (setf *print-array* nil)))



(defmethod (terrain :read-terrain-from-disk)
           (file-name)
  (with-open-file
    (input-file
      (merge-pathnames file-name "robot:kwak.robot.terrain-data;") :direction :input)
    (setf *print-array* t)
    (setf terrain-data (read input-file))
    (setf terrain-height-array (read input-file))
    (setf radius (read input-file))
    (setf alpha (read input-file))
    (setf beta (read input-file))
    (setf *print-array* nil)))



;*********************************************************

;  terrain.display-cursor

;*********************************************************



(defmethod (terrain :draw-cursor)
           ( position )
  (let* ((x (first position))
         (y (second position))
         (p1 (list (+ x 0.2) (+ y 0.2) 0))
         (p2 (list (+ x 0.8) (+ y 0.2) 0))
         (p3 (list (+ x 0.8) (+ y 0.8) 0))
         (p4 (list (+ x 0.2) (+ y 0.8) 0))
         (points (list p2 p3 p4 p1)))
    (move-to-earth p1)
    (do ((points points (cdr points)))
```

```
        ((null points) 'done-draw-cursor)
      (draw-to-earth (car points)))))


(defmethod (terrain :draw-obstacles)
           ()
  (dotimes (i max-x)
    (dotimes (j max-y)
      (cond ((equal 1 (aref terrain-data i j))
             (move-to-earth
               (list i j (aref terrain-height-array i)))
             (draw-to-earth
               (list (+ i 1) (+ j 1) (aref terrain-height-array (+ i 1))))
             (move-to-earth
               (list (+ i 1) j (aref terrain-height-array (+ i 1))))
             (draw-to-earth
               (list i (+ j 1) (aref terrain-height-array i)))))))


(defmethod (terrain :erase-cursor)
           ( position )
  (let* ((x (first position))
         (y (second position))
         (p1 (list (+ x 0.2) (+ y 0.2) 0))
         (p2 (list (+ x 0.8) (+ y 0.2) 0),
         (p3 (list (+ x 0.8) (+ y 0.8) 0))
         (p4 (list (+ x 0.2) (+ y 0.8) 0))
         (points (list p2 p3 p4 p1)))
    (move-to-earth p1)
    (do ((points points (cdr points)))
        ((null points) 'done-erase-cursor)
      (erase-to-earth (car points)))))


(defmethod (terrain :random-terrain)
           (obstacle-ratio random-seed)
  (let ((a 43411) (b 17) (c 640001)(percent nil) (seed nil) (x nil))
    (setf percent obstacle-ratio)
    (setf seed random-seed)
    (setf x seed)
    (dotimes (i max-x)
      (dotimes (j max-y)
    (if (< (/ (setf x (mod (+ (* a x) b) c)) c) (/ percent 100))
        (setf (aref terrain-data i j) 1)))))
  (send self :draw-obstacles))


(defmethod (terrain :add-ditch)
           (width location)
  (dotimes (i width)
    (dotimes (j max-y)
      (setf (aref terrain-data (+ i location) j) 1))))


;*****************************************************************
;
;  graph-vehicle flavor definition
;
```

```
;*********************************************************

(defflavor graph-vehicle((vehicle-points (make-array 28))
                         (body-points (make-array 10))
                         (polygons (make-array 13))
                         (numpolys nil)
                         (vertices (make-array 100)))
          ()
  :initable-instance-variables)



(defmethod (graph-vehicle :init-data)
           ()
  (send self :read-vehicle-data))  ; read data from disk


(defmethod (graph-vehicle :display)
           (a-H foot-positions)
  (clear-robot-window)
  (send self :body-pento-wrt-earth a-H foot-positions)
  (send self :draw-vehicle vehicle-points)
  (copy-terrain-to-robot-window)
  (make-visible))



(defmethod (graph-vehicle :read-vehicle-data)
           ()
; global variables : vehicle-points, polygons, numpolys, vertices
; format of file : num-of-points    num-of-polygons
;                  ( num a-vehicle-point) ....
;                  ( num-of-vertices  vertices-number-of-a-polygon)...
  (let* ((vehicle-file (open "exp3:kwak.robot;vehicle.data"))
     (numpts (read vehicle-file))
     (numvtces 0) (a-polygon nil))
    (setf numpolys (read vehicle-file))
    (dotimes (i numpts)
      (setf (aref vehicle-points i) (cdr (read vehicle-file))))
    (dotimes (i 10)
      (setf (aref body-points i) (aref vehicle-points i)))
    (dotimes (i numpolys)
      (setf a-polygon (read vehicle-file))
      (setf (aref polygons i) (list numvtces (car a-polygon)))
      (do ((a-polygon-vertices (cdr a-polygon) (cdr a-polygon-vertices))
       (j 0 (+ j 1)))
       ((null a-polygon-vertices))
    (setf (aref vertices  (+ numvtces j))
          (- (first a-polygon-vertices) 1)))
      (setf numvtces (+ numvtces (car a-polygon))))
    (close vehicle-file)))


(setf graph-asv (make-instance 'graph-vehicle))


;*********************************************************
;
;  graph-vehicle.display
```

```
;
;*********************************************************


(defmethod (graph-vehicle :body-pento-wrt-earth)
           ( a-H foot-positions )
; library : transform
  (let ((s1 0.6616) (s2 0.945) (s3 3.308) (l 0.8133) (m 1.0467)
;     (hipx-list '(5.1667 5.1667 0.0 0.0 -4.9167 -4.9167))
    (hipx-list '(6.0 6.0 0.0 0.0 -6.0 -6.0))
    (hipy-list '(1.62 -1.62 1.62 -1.62 1.62 -1.62))
    (sign1-list '(1 -1  1 -1  1 -1))
    (sign2-list '(1  1  1  1 -1 -1)))
   (send self :transform-body-points a-H body-points)
    (do ((positions foot-positions (cdr positions))
     (hipx-list hipx-list (cdr hipx-list))
     (hipy-list hipy-list (cdr hipy-list))
     (sign1-list sign1-list (cdr sign1-list))
     (sign2-list sign2-list (cdr sign2-list))
     (i 0 (+ i 1)))
    ((null positions) nil)
     (let* ((foot-pos (car positions))
         (hipx (car hipx-list)) (hipy (car hipy-list))
         (sign1 (car sign1-list)) (sign2 (car sign2-list))
         (px (- (first foot-pos) hipx))
         (py (- (second foot-pos) hipy))
         (pz (third foot-pos))
         (theta (vehicle-theta py pz m sign1))
         (dm (vehicle-dm px sign2))
         (dl (vehicle-dl py pz m l))
         (top-pos nil) (knee-pos nil))
    (setf top-pos
          (transform a-H
             (vehicle-top-pos hipx hipy m l dl theta sign1)))
    (setf knee-pos
          (transform a-H
             (vehicle-knee-pos hipx hipy m l s1 s2 s3
                       dl dm theta sign1 sign2)))
    (setf foot-pos (transform a-H foot-pos))
    (setf (aref vehicle-points (+ 10 (* 3 i)))
          top-pos)
    (setf (aref vehicle-points (+ 11 (* 3 i)))
          knee-pos)
    (setf (aref vehicle-points (+ 12 (* 3 i)))
          foot-pos)))))


(defmethod (graph-vehicle :draw-vehicle)
           ( vehicle-points )
; global variables :  polygons, numpolys, vertices
  (dotimes (i numpolys)
    (let ((start (first (aref polygons i)))
      (num-vertices (second (aref polygons i))))
      (move-to-earth (aref vehicle-points
               (aref vertices start)))
     (dotimes (j num-vertices)
       (draw-to-earth-d (aref vehicle-points
                (aref vertices (+ start j))))))
     )))
```

```
;*************************************************************
;
;  graph-vehicle.display.body-pento-wrt-earth
;
;*************************************************************


(defmethod (graph-vehicle :transform-body-points)
           (a-H body-points)
; globals : vehicle-points
; library : transform
  (dotimes (i 10)
    (setf (aref vehicle-points i)
      (transform a-H (aref body-points i)))))


(defun vehicle-dl (py pz m l)
  (/ (- (sqrt (+ (* py py) (* pz pz) (- (* m m)))) l)
     4))


(defun vehicle-dm (px sign2)
  (* sign2 (/ px 5)))


(defun vehicle-knee-pos (hipx hipy m l s1 s2 s3
                         dl dm theta sign1 sign2)
  (let* ((numer (+ (* s1 s1) (- (* s2 s2)) (* dl dl) (* dm dm)))
         (denom (* 2 s1 (sqrt (+ (* dl dl) (* dm dm)))))
         (beta (acos (/ numer denom)))
         (alpha (- (/ pi 2) (atan dm dl) beta))
         (sina (sin alpha)) (cosa (cos alpha))
         (sint (sin theta)) (cost (cos theta))
         (temp (- (* s3 sina) (- dl l)))
         (xk (+ (* sign2 s3 cosa) hipx))
         (yk (+ (* sign1 (+ (* temp sint) (* m cost))) hipy))
         (zk (- (+ (* temp cost) (* m sint)))))
    (list xk yk zk)))


(defun vehicle-theta (py pz m sign1)
  (let* ((angle1 (atan (* sign1 py) (* -1 pz)))
         (angle2 (atan m (sqrt (+ (* py py)
                                  (* pz pz)
                                  (- (* m m)))))))
    (- angle1 angle2)))


(defun vehicle-top-pos (hipx hipy m l dl theta sign1)
  (let* ((xt hipx)
         (l-dl (- l dl))
         (sina (sin theta))
         (cosa (cos theta))
         (yt (+ (* sign1 (+ (* m cosa) (* l-dl sina))) hipy))
         (zt (- (* m sina) (* l-dl cosa))))
    (list xt yt zt)))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************************
;
;  ditch-robot definition
;
;*****************************************************************

(defflavor ditch-robot()
           (test-overlap-robot)
  )


(defmethod (ditch-robot :initti)
           ()
  (send graph-asv :init-data)
  (setf vision-system (make-instance 'ditch-vision-system :owner self))
  (send vision-system :initti)
  (setf joystick (make-instance 'joystick))
  (send joystick :reset)
  (empty-queue lift-queue)
  (setf lift-flag t)
  (let ((H))
    (setf body (make-instance 'stop-body :owner self))
    (setf H (send body :initti))
    (setf legs (list
                 (make-instance 'test-overlap-leg :name 'leg1  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg2  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg3  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg4  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg5  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg6  :owner self)
                 ))
    (mapcar #'(lambda (a-leg) (send a-leg :initti H)) legs))
  )


(defmethod (ditch-robot :at-stability-limit)
           ()
  (not (send self :stable)))


(defmethod (ditch-robot :stop-motion)
           ()
  (send body :stop-body-motion))


(defmethod (ditch-robot :resume-motion)
           ()
  (send body :restore-body-motion)
  t)

;*****************************************************


(defun at_stability_limit ()
  (send asv :at-stability-limit))
```

```
(defun stop_motion ()
  (send asv :stop-motion))

(defun resume_motion ()
  (send asv :resume-motion))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************************
;
;  ditch-vision-system definition
;
;*****************************************************************


(defflavor ditch-vision-system ()
           (vision-system)
  )



(defmethod (ditch-vision-system :on-ditch-area)
           (body-H10)
  (let ((x (aref body-H10 0 3)))
    (cond ((and (>= x (- 21 7))
                (<= x (+ 21 *ditch-width*)))
           t)
          (t nil))))


(setf *ditch-width* 6)
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************************
;
;  executor flavor definition
;
;*****************************************************************


(defflavor executor
        (leg-pos-wrt-body desired-foothold-pos-wrt-earth
                          time command owner sensor (lift-height 1.4)
                          (T1 0.6) (T2 1.0) (T3 0.4) (T4 0.6)
                          (planned-contact-time 0.4) self-time
                          (sampling-time 0.1) ready-pos
                          H1 inv-H1 body-trans-rate1 body-rotate-rate1)
        ()
  :initable-instance-variables)


(defmethod (executor :set-desired-pos)
           (a-pos)
  (setf desired-foothold-pos-wrt-earth a-pos))



(defmethod (executor :get-desired-pos)
           ()
  desired-foothold-pos-wrt-earth)


(defmethod (executor :send-command)
           (a-command)
  (setf command a-command))


(defmethod (executor :set-time)
           (a-time)
  (setf time a-time))


(defmethod (executor :leg-pos-wrt-body)
           ()
  leg-pos-wrt-body)


(defmethod (executor :move)
           (H inv-H body-trans-rate body-rotate-rate)
  (setf H1 H)
  (setf inv-H1 inv-H)
  (setf body-trans-rate1 body-trans-rate)
  (setf body-rotate-rate1 body-rotate-rate)
  (cond ((equal command 'ready)
         (send self :move-in-ready))
        ((equal command 'advance)
         (send self :move-in-advance))
        ((equal command 'descent)
         (send self :move-in-descent))
```

```
        ((equal command 'contact)
         (send self :move-in-contact))
        ((equal command 'support)
         (send self :move-in-support))
        ((equal command 'lift)
         (send self :move-in-lift))
        ((equal command 'return)
         (send self :move-in-return))
        )
  )


(defmethod (executor :move-in-contact)
           ()
  (let ((leg-velocity-wrt-body (send self :find-velocity-wrt-body)))
    (setf leg-pos-wrt-body
          (vectadd (magvect sampling-time leg-velocity-wrt-body)
                   leg-pos-wrt-body))))


(defmethod (executor :find-velocity-wrt-body)
           ()
; returns foot-velocity-wrt-body
; velocity = - ( body-trans-rate + body-rotate-rate X leg-pos )
; globals v : body-trans-rate1, body-rotate-rate1
; lib : vectsub, vectadd, crossprod
  (vectsub '(0 0 0)
           (vectadd body-trans-rate1
                    (crossprod body-rotate-rate1 leg-pos-wrt-body))))


(defmethod (executor :move-in-advance)
           ()
  (let ((desired-pos (send self :desired-advance-pos-wrt-body))
        (dt (- T1 time)))
    (send self :move-del desired-pos leg-pos-wrt-body dt))
  (setf self-time 0.0))


(defmethod (executor :desired-advance-pos-wrt-body)
           ()
; a-pos is desired-stepping-pos-wrt-earth
; returns desired-pos-wrt-body in deploy state
; global variable : H1, inv-H1
; global function : to-earth-transform, to-body-transform, find-terrain-hegiht
  (let* ((desired-pos-wrt-earth desired-foothold-pos-wrt-earth)
         (terrain-height (third (send owner :terrain-point desired-pos-wrt-earth)))
         (desired-pos-height-wrt-earth (+ terrain-height lift-height))
         (pos-wrt-earth (list (first  desired-pos-wrt-earth)
                              (second desired-pos-wrt-earth)
                               desired-pos-height-wrt-earth)))
    (to-body-transform inv-H1 pos-wrt-earth)))


(defmethod (executor :move-in-descent)
           ()
; global function : to-body-transform
; global variables : inv-H1
  (let ((dt (- planned-contact-time self-time)))
    (if (< dt 0.05)
```

```
        (setf leg-pos-wrt-body (to-body-transform
                                  inv-H1 desired-foothold-pos-wrt-earth))
        (send self :move-del
          (to-body-transform inv-H1 desired-foothold-pos-wrt-earth)
          leg-pos-wrt-body dt)))
;  )


;(defmethod (executor :move-in-descent :after)
;           ()
  (setf self-time (+ self-time sampling-time)))


(defmethod (executor :move-del)
           (desired-pos present-pos dt)
; set new leg-pos depending on the arguments
; lib : vectadd, magvect
  (if (< dt 0.05)
      (setf leg-pos-wrt-body desired-pos)
      (let* ((inv-time-diff (/ 1 dt))
             (del (vectsub desired-pos present-pos))
             (velocity (magvect inv-time-diff del)))
        (setf leg-pos-wrt-body
              (vectadd present-pos (magvect sampling-time velocity))))))


(defmethod (executor :move-in-lift)
           ()
  (let* ((dt (- T3 time))
         (desired-pos (send self :lift-pos-desired))
         (z (third desired-pos)))
    (send self :move-del desired-pos leg-pos-wrt-body  dt)
    (setf ready-pos
          (list (first ready-pos) (second ready-pos) z))))


(defmethod (executor :lift-pos-desired)
; returens position-wrt-body which will be at the end of lift state.
; global f : to-body-transform,
; global v : inv-H1
           ()
  (let* ((leg-pos-wrt-earth (to-earth-transform H1 leg-pos-wrt-body))
         (desired-height (+ lift-height (third (send owner :terrain-point leg-pos-wrt-ea
h)))))
    (to-body-transform inv-H1 (list (first  leg-pos-wrt-earth)
                                    (second leg-pos-wrt-earth)
                                    desired-height))))


(defmethod (executor :move-in-ready)
           ()
  (setf leg-pos-wrt-body ready-pos))


(defmethod (executor :move-in-return)
           ()
; Modifying leg-pos-z is redundent but it can correct disturbance by itself.
  (let ((dt (- T4 time))
```

```
        (desired-pos ready-pos))
    (send self :move-del desired-pos leg-pos-wrt-body dt)))


(defmethod (executor :move-in-support)
           ()
; globals : body-trans-rate1, body-rotate-rate1
; lib : vectadd, magvect
; In general terrain, leg-pos-z should be updated by real terrain height.
  (let ((leg-velocity-wrt-body (send self :find-velocity-wrt-body)))
    (setf leg-pos-wrt-body
          (vectadd (magvect sampling-time leg-velocity-wrt-body)
                   leg-pos-wrt-body))))


(defmethod (executor :initti)
           (leg-name init-H)
  (setf sensor (send owner :contact-sensor))
  (let ((x (aref init-H 0 3))
        (y (aref init-H 1 3))
        (z (aref init-H 2 3)))
    (cond ((equal leg-name 'leg1)
           (setf ready-pos '( 5  3 -4))
           (setf leg-pos-wrt-body (list  6  3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 6) (+ y 3) 0)))
          ((equal leg-name 'leg2)
           (setf ready-pos '( 5 -3 -4))
           (setf leg-pos-wrt-body (list  5 -3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 5) (- y 3) 0)))
          ((equal leg-name 'leg3)
           (setf ready-pos '( 0  3 -4))
           (setf leg-pos-wrt-body (list  0  3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 0) (+ y 3) 0)))
          ((equal leg-name 'leg4)
           (setf ready-pos '( 0 -3 -4))
           (setf leg-pos-wrt-body (list  0 -3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 0) (- y 3) 0)))
          ((equal leg-name 'leg5)
           (setf ready-pos '(-5  3 -4))
           (setf leg-pos-wrt-body (list -5  3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (- x 5) (+ y 3) 0)))
          ((equal leg-name 'leg6)
           (setf ready-pos '(-5 -3 -4))
           (setf leg-pos-wrt-body (list -5 -3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (- x 5) (- y 3) 0)))))
  )
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************************
;
;  load file (f-load-t-444.lisp)
;
;*****************************************************************


;
; graph-terrain is used in sensor and vision
;
; Overlapped working volume (1 foot)
; Front and rear are not extended.
;
; logic change
;

(load "robot:kwak.robot;math-t")

(load "robot:kwak.robot;user-interface-t2")
(load "robot:kwak.robot;graph-t1")
(load "robot:kwak.robot;display-t2")

(load "robot:kwak.robot1;vision-t")
(load "robot:kwak.robot5;ditch-vision-t")

(load "robot:kwak.robot;tkm-t")
(load "robot:kwak.robot4;overlap-tkm-t")
(load "robot:kwak.robot1;foothold-t")
(load "robot:kwak.robot4;overlap-foothold-t")
(load "robot:kwak.robot;sensor-t")
(load "robot:kwak.robot1;executor-t")
(load "robot:kwak.robot;control-machine-t")
(load "robot:kwak.robot;plan-machine-t")
(load "robot:kwak.robot1;leg-t")
(load "robot:kwak.robot4;overlap-leg-t")
(load "robot:kwak.robot5;test-overlap-leg-t-441")

(load "robot:kwak.robot;stability-t2")
(load "robot:kwak.robot;support-plane-t")
(load "robot:kwak.robot;h-calculator-t")
(load "robot:kwak.robot;command-regulator-t")
(load "robot:kwak.robot;terrain-regulator-t")
(load "robot:kwak.robot;body-controller-t")
(load "robot:kwak.robot1;body-t")
(load "robot:kwak.robot6;stop-body-t")

(load "robot:kwak.robot1;robot-t1")
(load "robot:kwak.robot4;overlap-robot-t")
(load "robot:kwak.robot5;test-overlap-robot-t-442")
(load "robot:kwak.robot6;ditch-robot-t")

(load "robot:kwak.robot6;robot444")

(load "robot:kwak.robot;add-to-system-menu")

(setf asv (make-instance 'ditch-robot))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  foothold-finder definition
;
;*****************************************************************


(defflavor foothold-finder(sixteen-footholds
                           four-lines tkm-calculator
                           (no-cell-available-flag nil)
                           (TKM-margin 0.4) owner)
           ()
  :initable-instance-variables)


(defmethod (foothold-finder :initti)
           (leg-name)
  (cond ((equal leg-name 'leg1)
         (setf sixteen-footholds
               '(( 7.3  4.3) ( 7.3  3.3) ( 7.3  2.3) ( 7.3  1.3)
                 ( 6.3  4.3) ( 6.3  3.3) ( 6.3  2.3) ( 6.3  1.3)
                 ( 5.3  4.3) ( 5.3  3.3) ( 5.3  2.3) ( 5.3  1.3)
                 ( 4.3  4.3) ( 4.3  3.3) ( 4.3  2.3) ( 4.3  1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) ( 8.0832  2.7339 0))
                 ((0 -0.3420 -0.9397) ( 8.0832  2.7339 0))
                 ((0 -0.3420 -0.9397) ( 3.4167  2.7339 0))
                 ((0  0.3420 -0.9397) ( 3.4167  2.7339 0)))))
        ((equal leg-name 'leg2)
         (setf sixteen-footholds
               '(( 7.3 -4.3) ( 7.3 -3.3) ( 7.3 -2.3) ( 7.3 -1.3)
                 ( 6.3 -4.3) ( 6.3 -3.3) ( 6.3 -2.3) ( 6.3 -1.3)
                 ( 5.3 -4.3) ( 5.3 -3.3) ( 5.3 -2.3) ( 5.3 -1.3)
                 ( 4.3 -4.3) ( 4.3 -3.3) ( 4.3 -2.3) ( 4.3 -1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) ( 8.0832 -2.7339 0))
                 ((0 -0.3420 -0.9397) ( 8.0832 -2.7339 0))
                 ((0 -0.3420 -0.9397) ( 3.4167 -2.7339 0))
                 ((0  0.3420 -0.9397) ( 3.4167 -2.7339 0)))))
        ((equal leg-name 'leg3)
         (setf sixteen-footholds
               '(( 1.5  4.3) ( 1.5  3.3) ( 1.5  2.3) ( 1.5  1.3)
                 ( 0.5  4.3) ( 0.5  3.3) ( 0.5  2.3) ( 0.5  1.3)
                 (-0.5  4.3) (-0.5  3.3) (-0.5  2.3) (-0.5  1.3)
                 (-1.5  4.3) (-1.5  3.3) (-1.5  2.3) (-1.5  1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) ( 2.2915  2.7339 0))
                 ((0 -0.3420 -0.9397) ( 2.2915  2.7339 0))
                 ((0 -0.3420 -0.9397) (-2.2915  2.7339 0))
                 ((0  0.3420 -0.9397) (-2.2915  2.7339 0)))))
        ((equal leg-name 'leg4)
         (setf sixteen-footholds
               '(( 1.5 -4.3) ( 1.5 -3.3) ( 1.5 -2.3) ( 1.5 -1.3)
                 ( 0.5 -4.3) ( 0.5 -3.3) ( 0.5 -2.3) ( 0.5 -1.3)
                 (-0.5 -4.3) (-0.5 -3.3) (-0.5 -2.3) (-0.5 -1.3)
                 (-1.5 -4.3) (-1.5 -3.3) (-1.5 -2.3) (-1.5 -1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) ( 2.2915 -2.7339 0))
                 ((0 -0.3420 -0.9397) ( 2.2915 -2.7339 0))
                 ((0 -0.3420 -0.9397) (-2.2915 -2.7339 0))
                 ((0  0.3420 -0.9397) (-2.2915 -2.7339 0)))))
```

```
        ((equal leg-name 'leg5)
         (setf sixteen-footholds
               '((-4.0  4.3) (-4.0  3.3) (-4.0  2.3) (-4.0  1.3)
                 (-5.0  4.3) (-5.0  3.3) (-5.0  2.3) (-5.0  1.3)
                 (-6.0  4.3) (-6.0  3.3) (-6.0  2.3) (-6.0  1.3)
                 (-7.0  4.3) (-7.0  3.3) (-7.0  2.3) (-7.0  1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) (-3.3332  2.7339 0))
                 ((0 -0.3420 -0.9397) (-3.3332  2.7339 0))
                 ((0 -0.3420 -0.9397) (-7.8332  2.7339 0))
                 ((0  0.3420 -0.9397) (-7.8332  2.7339 0)))))
        ((equal leg-name 'leg6)
         (setf sixteen-footholds
               '((-4.0 -4.3) (-4.0 -3.3) (-4.0 -2.3) (-4.0 -1.3)
                 (-5.0 -4.3) (-5.0 -3.3) (-5.0 -2.3) (-5.0 -1.3)
                 (-6.0 -4.3) (-6.0 -3.3) (-6.0 -2.3) (-6.0 -1.3)
                 (-7.0 -4.3) (-7.0 -3.3) (-7.0 -2.3) (-7.0 -1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) (-3.3332 -2.7339 0))
                 ((0 -0.3420 -0.9397) (-3.3332 -2.7339 0))
                 ((0 -0.3420 -0.9397) (-7.8332 -2.7339 0))
                 ((0  0.3420 -0.9397) (-7.8332 -2.7339 0)))))

        )
  (setf tkm-calculator (send owner :tkm-calculator))
  )




(defmethod (foothold-finder :find-foothold)
           (H6 inv-H6 body-trans-rate10 body-rotate-rate10
               estimated-support-plane)
; returns ((max-foothold max-tkm) (foothold-list) (tkm-list))
; all points are wpt body coordinate system.
  (let* ((estimated-support-plane-wrt-body
           (plane-transform estimated-support-plane H6))
         (four-points (send self
                            :four-points-on-support-plane
                            four-lines estimated-support-plane-wrt-body))
         (possible-footholds (send self
                                   :get-possible-footholds
                                   (send self
                                     :estimate-footholds
                                     four-points estimated-support-plane-wrt-body)
                                   H6 inv-H6)))
    (send self
      :get-foothold-with-max-TKM
      possible-footholds H6
      body-trans-rate10 body-rotate-rate10)))



;"*****************************************************"

; foothold-finder.find-foothold

;"*****************************************************"
```

```
(defmethod (foothold-finder :estimate-footholds)
           (four-points-wrt-body estimated-support-plane-wrt-body)
; returns estimate-footholds-wrt-body
  (do* ((footholds sixteen-footholds (cdr footholds))
        (out-footholds nil)
        (a-foothold nil))
       ((null footholds)
        (get-points-on-support-plane out-footholds estimated-support-plane-wrt-body))
    (setf a-foothold (car footholds))
    (if (in-side-of-polygon a-foothold
                            (pick-two-dimensions four-points-wrt-body))
        (setf out-footholds (cons a-foothold out-footholds)))))


(defmethod (foothold-finder :four-points-on-support-plane)
           (four-lines estimated-support-plane-wrt-body)
; returns four points which are intersected by four-lines on
; estimated-support-plane-wrt-body
; math lib: plane-intersection
  (do* ((lines four-lines (cdr lines))
        (points nil))
       ((null lines) points)
    (setf points (cons (plane-intersection (car lines)
                                           estimated-support-plane-wrt-body)
                       points))))


(defmethod (foothold-finder :get-foothold-with-max-TKM)
           (possible-footholds H
                               body-trans-rate body-rotate-rate)
; returns ((max-foothold max-tkm) (foothold-list) (tkm-list))
; sets no-cell-available-flag
; real-footholds is really possible footholds
    (do ((footholds possible-footholds (cdr footholds))
         (max-foothold nil) (a-foothold nil) (TKM-list nil) (a-TKM nil)
         (real-footholds nil) (max-TKM -100.0))
        ((null footholds)
         (setf no-cell-available-flag (< max-TKM TKM-margin))
         (if (>= max-TKM TKM-margin)
             (make-output-form
               max-foothold max-TKM real-footholds TKM-list H)
             nil))
      (setf a-foothold (car footholds))
      (setf a-TKM (send tkm-calculator :find-tkm
                        a-foothold body-trans-rate body-rotate-rate))
      (if a-TKM
          (progn (setf TKM-list (cons a-TKM TKM-list))
                 (setf real-footholds (cons a-foothold real-footholds))
                 (if (> a-TKM max-TKM)
                     (progn (setf max-TKM a-TKM)
                            (setf max-foothold a-foothold)))))))


(defmethod (foothold-finder :get-possible-footholds)
           (estimated-footholds H inv-H)
; returns possible-footholds wrt body
  (co-body-transform inv-H
                     (send self :find-possible-footholds
                           (to-earth-transform H estimated-footholds)))
```

```
  ))



;"*****************************************************"

;  foothold-finder.estimate-foothold

;"*****************************************************"


(defun check-polarity (point1 point2 point3)
  (let* ((vect1 (vectsub point2 point1))
         (vect2 (vectsub point3 point1)))
    (if (not (third vect1))
        (progn (setf vect1 (reverse (cons 0 (reverse vect1))))
               (setf vect2 (reverse (cons 0 (reverse vect2))))))
    (crossprod vect1 vect2)))


(defun get-points-on-support-plane (points estimated-support-plane-wrt-body)
; returns intersection points with support plane in z-body direction.
; math lib: plane-intersection
  (do* ((points points (cdr points))
        (out-points nil))
       ((null points) out-points)
    (setf out-points (cons (plane-intersection
                              (make-line-to-get-point-on-support-plane
                                  (car points))
                              estimated-support-plane-wrt-body) out-points))))


(defun in-side-of-polygon (a-point polygon-points)
; polygon-points must be convext-polygon and in order & two dimensional points.
  (do* ((first-points  polygon-points (cdr first-points))
        (second-points (reverse (cons (car first-points)
                                      (reverse (cdr first-points))))
                       (cdr second-points))
        (signs nil) (first-point nil) (second-point nil))
       ((null first-points) (same-polarity signs))
    (setf first-point  (car first-points))
    (setf second-point (car second-points))
    (setf signs (cons (check-polarity first-point second-point a-point)
                      signs))))


(defun make-line-to-get-point-on-support-plane (a-point)
; a-point is two dimensional point.
; returns a-line ((z-direction) (a-point -100))
  (list '(0 0 1) (list (first a-point) (second a-point) -100)))


(defun pick-two-dimensions (points)
  (if (listp (first points))
      (do* ((points* points (cdr points))        ; more than one point case
            (a-point nil)
            (out-points nil))
```

```
          ((null points) out-points)
        (setf a-point (car points))
        (setf out-points (cons (list (first a-point) (second a-point))
                               out-points)))
      (list (first points) (second points))))  ; one point case


(defun same-polarity (signs)
  (do ((signs (cdr signs) (cdr signs))
       (first-sign (plusp (third (car signs))))
       (same T))
      ((null signs) same)
    (if (not (equal first-sign (plusp (third (car signs)))))
        (setf same nil))))




;"*****************************************************"

;  foothold-finder.find-foothold.get-foothold-with-MAX-tkm

;"*****************************************************"


(defun make-output-form
       (max-foothold max-TKM possible-footholds TKM-list H)
; output-form : ((foothold-with-max-tkm tkm)
;                (leg-projected-permitted-footholds)
;                (leg-projected-TKM-list))
; output footholds are in earth coordinate.
; math lib : to-earth-transform
  (list (list (to-earth-transform H max-foothold) max-TKM)
        (to-earth-transform H possible-footholds)
        TKM-list))




;"*****************************************************"

;  foothold-finder.select-foothold.get-possible-foothold

;"*****************************************************"


(defmethod (foothold-finder :find-possible-footholds)
           (estimated-footholds-wrt-earth)
; returns possible-footholds-wrt-earth
; graph-terrain is object.
  (do* ((footholds estimated-footholds-wrt-earth (cdr footholds))
        (a-foothold nil) (t-cell nil) (out-footholds nil))
       ((null footholds) (unique-footholds-only out-footholds))
    (setf a-foothold (car footholds))
    (setf t-cell (get-center-of-digitized-terrain-cell a-foothold))
;    (setf out-footholds
;          (cons (list (+ (first t-cell) 0.5)
```

```
;                (+ (second t-cell) 0.5)
;                0.0) out-footholds))))

    (if (send owner :permitted-cell t-cell)
        (setf out-footholds
              (cons (send owner :terrain-point t-cell)
                    out-footholds)))))


(defun get-center-of-digitized-terrain-cell (a-foothold)
; cell resolution is 1 foot by 1 foot
  (list (+ (floor (first  a-foothold)) 0.5)
        (+ (floor (second a-foothold)) 0.5)))


(defun unique-footholds-only (mixed-footholds)
  (do* ((footholds mixed-footholds (cdr footholds))
        (out-footholds nil)
        (a-foothold nil))
       ((null footholds) out-footholds)
    (setf a-foothold (car footholds))
    (if (not (member a-foothold out-footholds :test 'equal))
        (setf out-footholds (cons a-foothold out-footholds)))))
```

```
;;; -*- Mode:Common-Lisp; Package:USER; Base:10 -*-

;*********************************************************
;
;  low level graph routines
;
;*********************************************************


(defvar *robot-display-window* nil)
(defvar *robot-display-window-array* nil)
(defvar *robot-window* nil)
(defvar *robot-window-array* nil)
(defvar *robot-window-width* nil)
(defvar *robot-window-height* nil)
(defvar *terrain-buffer* nil)
(defvar *terrain-buffer-array* nil)
(defvar *max-y* nil)
(defvar *start-point* nil)
;TI
(defvar *xs* (make-array 2))
(defvar *ys* (make-array 2))


(defun copy-terrain-to-robot-window ()
  (tv:sheet-force-access (*robot-window*)
    (send *robot-window* :bitblt
      tv:alu-ior *robot-window-width* *robot-window-height*
      *terrain-buffer-array* 2 2 0 0)))


(defun draw-to (a-point a-window)
; global variables : *start-point*
  (tv:sheet-force-access (a-window)
    (send a-window ':draw-line (first *start-point*)
      (- *max-y* (second *start-point*))
      (first a-point)
      (- *max-y* (second a-point))   tv:alu-ior))
  (setq *start-point* a-point))


(defun draw-to-d (a-point a-window)
; global variables : *start-point*
  (tv:sheet-force-access (a-window)
      (setf (aref *xs* 0) (+ 4 (first *start-point*)))
      (setf (aref *xs* 1) (+ 4 (first a-point)))
      (setf (aref *ys* 0) (+ 4 (- *max-y* (second *start-point*))))
      (setf (aref *ys* 1) (+ 4 (- *max-y* (second a-point))))
    (send a-window :draw-wide-curve *xs* *ys* 2))
  (setq *start-point* a-point))


(defun erase-to (a-point a-window)
; global variables : *start-point*
  (tv:sheet-force-access (a-window)
    (send a-window ':draw-line (first *start-point*)
      (- *max-y* (second *start-point*))
      (first a-point)
      (- *max-y* (second a-point))   tv:alu-andca))
  (setq *start-point* a-point))
```

```
(defun get-keyboard-input()
; This is not for the graphics, but this function uses Zeta LISP.
; This is the reason why this function is in Zeta graphic package.
  (send *terminal-io* :tyi-no-hang))


(defun make-robot-window ()
  (setq *robot-display-window* (tv:make-window
                                 'tv:window
                                 :blinker-p nil
                                 :position '(0 0)
                                 :width *screen-width*
                                 :height (truncate (* 0.8 *screen-height*))
                                 :borders 2
                                 :label "robot-display-window"
                                 :name "robot-display-window"
                                 :save-bits t
                                 :expose-p t))
  (let* ((r-w (send *robot-display-window* :width))
         (r-h (send *robot-display-window* :height))
         (r-x nil) (r-y nil))
    (multiple-value (r-x r-y) (send *robot-display-window* :position))
    (setq *robot-window* (tv:make-window '
                          tv:window
                          :position (list r-x r-y)
                          :width r-w
                          :height r-h
                          :blinker-p nil
                          :borders 2
                          :label "robot-window"
                          :name "robot-window"
                          :save-bits t
                          :expose-p nil))
    (setq *terrain-buffer* (tv:make-window
                            'tv:window
                            :position (list r-x r-y)
                            :width r-w
                            :height r-h
                            :blinker-p nil
                            :borders 2
                            :label "terrain-buffer"
                            :name "terrain-buffer"
                            :save-bits t
                            :expose-p nil))
    (setq *max-y* (send *robot-window* :inside-height)))
  (setq *robot-display-window-array* (send *robot-display-window* :bit-array))
  (setq *robot-window-array* (send *robot-window* :bit-array))
  (setq *robot-window-width* (send *robot-window* :inside-width))
  (setq *robot-window-height* (send *robot-window* :inside-height))
  (setq *terrain-buffer-array* (send *terrain-buffer* :bit-array)))


(defun make-visible ()
  (send *robot-display-window* :bitblt
    tv:alu-seta *robot-window-width* *robot-window-height*
    *robot-window-array* 2 2 0 0))
```

```
(defun move-to (a-point)
; global variables : *start-point*
; This function just changes *start-point*.
  (setq *start-point* a-point))


(defun save-terrain-to-terrain-buffer()
  (tv:sheet-force-access (*terrain-buffer*)
    (send *terrain-buffer* :bitblt
      tv:alu-seta *robot-window-width* *robot-window-height*
      *robot-window-array* 2 2 0 0)))


(defun clear-robot-window ()
  (tv:sheet-force-access (*robot-window*)
    (send *robot-window* :clear-window)))


(defun middle-of-robot-window ()
  (list (/ (send *robot-window* :inside-width) 2)
        (/ (send *robot-window* :inside-height) 2)))


(defun kill-robot-terrain-windows()
  (send *robot-display-window* :kill)
  (send *robot-window* :kill)
  (send *terrain-buffer* :kill))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  H-calculator definition
;
;*****************************************************************


(defflavor H-calculator((sampling-time 0.1) H
                        old-H)
           ()
  :initable-instance-variables)


(defmethod (H-calculator :initti)
           ()
; library fucntion : ident
  (setf H  (ident))
  (setf (aref H 0 3) 6.5)
  (setf (aref H 1 3) 19.5)
  (setf (aref H 2 3) 5.4)
  H)


(defmethod (H-calculator :new-H)
           (body-trans-rate body-rotate-rate)
  (setf H
        (orthogonalization
          (get-new-H
            H
            (get-del-H
              H
              (get-delta body-trans-rate body-rotate-rate sampling-time))))))


(defmethod (H-calculator :save)
           ()
  (setf old-H H))


(defmethod (H-calculator :restore)
           ()
  (setf H old-H))



;"*****************************************************"

;  H-calculator.new-H

;"*****************************************************"


(defun get-delta (body-trans-rate body-rotate-rate sampling-time)
  (let* ((del-trans-x  (* (first  body-trans-rate)  sampling-time))
         (del-trans-y  (* (second body-trans-rate)  sampling-time))
         (del-trans-z  (* (third  body-trans-rate)  sampling-time))
         (del-rotate-x (* (first  body-rotate-rate) sampling-time))
         (del-rotate-y (* (second body-rotate-rate) sampling-time))
```

```
        (del-rotate-z (* (third  body-rotato-rate) sampling-time)))
    (list (list del-trans-x  del-trans-y  del-trans-z)
          (list del-rotate-x del-rotate-y del-rotate-z))))


(defun get-del-H (H delta-trans-rotate)
; math lib : ident
  (let* ((H-del (ident))        ; initialze identity matirix
         (delta-trans  (first  delta-trans-rotate))
         (delta-rotate (second delta-trans-rotate)))
    (setf (aref H-del 0 0) 0)
    (setf (aref H-del 1 0) (third delta-rotate))
    (setf (aref H-del 2 0) (- (second delta-rotate)))
    (setf (aref H-del 0 1) (- (third delta-rotate)))
    (setf (aref H-del 1 1) 0)
    (setf (aref H-del 2 1) (first delta-rotate))
    (setf (aref H-del 0 2) (second delta-rotate))
    (setf (aref H-del 1 2) (- (first delta-rotate)))
    (setf (aref H-del 2 2) 0)
    (setf (aref H-del 0 3) (first  delta-trans))
    (setf (aref H-del 1 3) (second delta-trans))
    (setf (aref H-del 2 3) (third  delta-trans))
    (setf (aref H-del 3 3) 0)
    (matrixmult H H-del)))


(defun get-new-H (H del-H)
  (matrixadd H del-H))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************************
;
;  leg flavor definition
;
;*****************************************************************


(defflavor leg (name owner plan-machine control-machine
                     executor contact-sensor tkm-calculator
                     foothold-finder exchanged-leg
                     foothold tkm foothold-list tkm-list tkm-p
                     reserved-foothold reserved-tkm)
           ()
  :initable-instance-variables
  :gettable-instance-variables)


(defmethod (leg :initti)
           (H)
    (setf contact-sensor  (make-instance 'contact-sensor  :owner self))
    (setf executor        (make-instance 'executor        :owner self))
    (setf control-machine (make-instance 'control-state-machine :owner self))
    (setf plan-machine    (make-instance 'plan-state-machine    :owner self))
    (setf tkm-calculator  (make-instance 'tkm-calculator  :owner self))
    (setf foothold-finder (make-instance 'foothold-finder :owner self))
    (setf foothold (send executor :initti name H))
    (send contact-sensor :initti name)
    (send control-machine :initti name)
    (send plan-machine :initti name)
    (send tkm-calculator :initti name)
    (send foothold-finder :initti name))


(defmethod (leg :contact-confirm)
           ()
  (send contact-sensor :contact-p))


(defmethod (leg :do-planned-motion)
           ()
  (send plan-machine :change)
  (send control-machine :change)
  (send executor :move (send owner :get-H1) (send owner :get-inv-H1)
        (send owner :get-body-trans-rate1)
        (send owner :get-body-rotate-rate1))
  (send contact-sensor :sensing))


(defmethod (leg :get-H1)
           ()
  (send owner :get-H1))


(defmethod (leg :has-foothold-p)
           ()
  foothold)
```

```
(defmethod (leg :interlock-confirm)
           ()
; may add stable-without-p self
  (if (send exchanged-leg :contact-confirm)
      t
      nil))


(defmethod (leg :leg-pos-wrt-body)
           ()
  (send executor :leg-pos-wrt-body))


(defmethod (leg :lift-able)
           ()
  (if (equal (send plan-machine :state-name) 'eligible-to-lift)
      self
      nil))

(defmethod (leg :lift-ok)
           ()
  (send owner :lift-ok name))


(defmethod (leg :lifted)
           ()
  (send owner :lifted name))


(defmethod (leg :new-foothold)
           ()
  (cond ((car foothold-list)
         (send self :set-max)
         t)
        (t
         nil)))


(defmethod (leg :permitted-cell)
           (t-cell)
  (send owner :permitted-cell t-cell))


(defmethod (leg :place-able)
           ()
; check plan state as well as foothold for the leg
  (if (equal (send plan-machine :state-name) 'available-leg)
      self
      nil))


(defmethod (leg :projected-pos)
           ()
```

```
  (send executor :get-desired-pos))


(defmethod (leg :select-foothold)
           ()
; out-list: ((max-foothold max-tkm) (foothold-list) (tkm-list))
  (let* ((H (send owner :get-H6))
         (inv-H (send owner :get-inv-H6))
         (body-trans-rate (send owner :get-body-trans-rate10))
         (body-rotate-rate (send owner :get-body-rotate-rate10))
         (estimated-support-plane
           (send owner :get-estimated-support-plane))
         (out-list
           (send foothold-finder :find-foothold
                                 H inv-H body-trans-rate body-rotate-rate
                                 estimated-support-plane)))
    (setf foothold (first (first out-list)))
    (setf reserved-foothold foothold)
    (setf tkm (second (first out-list)))
    (setf reserved-tkm tkm)
    (setf foothold-list (second out-list))
    (setf tkm-list (third out-list))))


(defmethod (leg :send-decision)
           (a-decision)
  (send plan-machine :send-decision a-decision)
;  )


;(defmethod (leg :send-decision :after)
;          (a-decision)
  (if (equal a-decision 'place)
      (send executor :set-desired-pos foothold)))


(defmethod (leg :send-exchange)
           (a-leg)
  (setf exchanged-leg a-leg))


(defmethod (leg :set-max)
           ()
  (do ((footholds (cdr foothold-list) (cdr footholds))
       (tkms (cdr tkm-list) (cdr tkms))
       (max-foothold (car foothold-list))
       (max-tkm (car tkm-list))
       (out-footholds) (out-tkms))
      ((null footholds)
       (setf foothold max-foothold)
       (setf tkm max-tkm)
       (setf foothold-list out-footholds)
       (setf tkm-list out-tkms))
    (cond ((> (car tkms) max-tkm)
           (setf max-foothold (car footholds))
           (setf max-tkm (car tkms)))
          (t
           (setf out-footholds
                 (cons (car footholds) out-footholds))
```

```
          (setf out-tkms
                (cons (car tkms) out-tkms))))))


(defmethod (leg :stable-without-p)
           ()
  (send owner :stable-without-p self))


(defmethod (leg :supporting)
           ()
  (cond ((equal (send plan-machine :state-name) 'planned-contact)
         self)
        ((equal (send plan-machine :state-name) 'eligible-to-lift)
         self)
        (t nil))
  )


(defmethod (leg :supporting-p)
           ()
  (cond ((equal (send control-machine :state-name) 'contact)
         self)
        ((equal (send control-machine :state-name) 'support)
         self)
        (t nil))
  )


(defmethod (leg :terrain-point)
           (t-cell)
  (send owner :terrain-point t-cell))


(defmethod (leg :TKM-limit)
           ()
  (cond ((null tkm)
         self)
        ((< tkm 0.1)
         self)
        (t
         nil)))


(defmethod (leg :TKM-limit-p)
           ()
  (cond ((null tkm-p)
         self)
        ((< tkm-p 0.5)
         self)
        (t nil)))


(defmethod (leg :update-tkm)
           ()
```

```
  (let ((body-trans-rate (send owner :get-body-trans-rate10))
        (body-rotate-rate (send owner :get-body-rotate-rate10))
        (inv-H (send owner :get-inv-H10)))
    (setf tkm (send tkm-calculator :find-tkm
                        (to-body-transform inv-H foothold)
                        body-trans-rate body-rotate-rate)))
    )


(defmethod (leg :update-tkm-p)
           ()
  (let ((body-trans-rate-p (send owner :get-body-trans-rate1))
        (body-rotate-rate-p (send owner :get-body-rotate-rate1))
        (inv-H-p (send owner :get-inv-H1)))
    (setf tkm-p (send tkm-calculator :find-tkm
                          (to-body-transform inv-H-p foothold)
                          body-trans-rate-p body-rotate-rate-p)))
  )


(defmethod (leg :with-foothold)
           ()
  (cond (reserved-foothold
         (setf foothold reserved-foothold)
         (setf tkm reserved-tkm)
         self)
        (t nil)))
```

```
;;; -*- Mode:Common-Lisp; Package:USER; Base:10 -*-

;*****************************************************************
;
;  robot math library
;
;*****************************************************************


(defun arc-cos (s)
  (acos s))


(defun col-mul(mat col1 col2)
  (let ((sum 0))
    (dotimes (i 4)
      (setf sum (+ sum (* (aref mat i col1) (aref mat i col2)))))
  sum))


(defun counting(a-list)
  (do ((a-list a-list (cdr a-list))
       (i 0 (+ i 1)))
      ((null a-list) i)))


(defun crossprod (vect1 vect2)
  (let* ((x1 (first  vect1)) (x2 (first  vect2))
         (y1 (second vect1)) (y2 (second vect2))
         (z1 (third  vect1)) (z2 (third  vect2))
         (x (- (* y1 z2) (* y2 z1)))
         (y (- (* x2 z1) (* x1 z2)))
         (z (- (* x1 y2) (* x2 y1))))
    (list x y z)))


(defun delete-list (a-list b-list)   ; delete a-list from b-list
  (do  ((deleting-list a-list (cdr deleting-list))
        (deleted-list b-list))
      ((null deleting-list) deleted-list)
    (setf deleted-list (remove (car deleting-list)
                               deleted-list :test 'equal))))


(defmacro dequeue (queue)
  `(prog1 (car ,queue)
          (setf ,queue (cdr ,queue))))


(defun dotprod (vect1 vect2)
; No dimension limitation !!!
  (apply '+ (mapcar '* vect1 vect2)))


(defmacro enqueue (queue-name element)
; globals : queue-name
```

```
; Value of recover field of command is a list.
; Two recover command is possible for one sampling-time.
; structure of QUEUE :  (first second third ... last)
  `(setq ,queue-name (nconc ,queue-name (list ,element))))


(defmacro empty-queue (queue)
  `(setq ,queue '()))


(defun ident()
  (make-array '(4 4):initial-contents
              '((1 0 0 0)
                (0 1 0 0)
                (0 0 1 0)
                (0 0 0 1))))


(defun magnitude (a-vector)
  (sqrt (dotprod a-vector a-vector)))


(defun magvect (const vect)
; magvect = const * vect
  (mapcar #'(lambda (a-element)
              (* const a-element))
          vect))


(defun matrixadd (mt1 mt2)
  (let ((mt3 (ident)))
    (dotimes (i 4)
      (dotimes (j 4)
        (setf (aref mt3 i j) (+ (aref mt1 i j) (aref mt2 i j)))))
    mt3))


(defun matrixinv(mat)
  (let ((px  (- (col-mul mat 0 3)))
        (py  (- (col-mul mat 1 3)))
        (pz  (- (col-mul mat 2 3)))
        (matrix (transpose mat)))
    (setf (aref matrix 3 0) 0)  (setf (aref matrix 3 1) 0)
    (setf (aref matrix 3 2) 0)  (setf (aref matrix 3 3) 1)
    (setf (aref matrix 0 3) px) (setf (aref matrix 1 3) py)
    (setf (aref matrix 2 3) pz)
    matrix))


(defun matrixmult (mt1 mt2)
  (let ((mat (make-array '(4 4)))) ;it defines 0 through 3. (4 is not included)
    (dotimes (i 4)      ; will repeat i=0, 1, 2, and 3. (not 4)
      (dotimes (j 4)
        (setf (aref mat i j) 0)         ; initialize to zero
        (dotimes (k 4)
          (setf (aref mat i j) (+ (aref mat i j) (* (aref mt1 i k)
```

```
                                                    (aref mt2 k j)))))))
    mat))


(defun nil-list(a-list)
  (do ((a-list a-list (cdr a-list))
       (not-nil nil))
      ((null a-list) (not not-nil))
    (if (car a-list)
        (setf not-nil t))))


(defun normalize-vector (a-vector)
  (let ((m (magnitude a-vector)))
    (if (< m 0.0000001)
        (list 0 0 0)
        (magvect (/ 1.0 m) a-vector))))


(defun orthogonalization (mt)
                ; Gram-Schimit orthogonalization process
  (let* ((mx (ident))
         (tx (aref mt 0 3)) (ty (aref mt 1 3)) (tz (aref mt 2 3))

         (x1 (aref mt 0 0)) (x2 (aref mt 0 1)) (x3 (aref mt 0 2))
         (y1 (aref mt 1 0)) (y2 (aref mt 1 1)) (y3 (aref mt 1 2))
         (z1 (aref mt 2 0)) (z2 (aref mt 2 1)) (z3 (aref mt 2 2))
         (m1 (magnitude (list x1 y1 z1)))
         (x1 (/ x1 m1))
         (y1 (/ y1 m1))
         (z1 (/ z1 m1))
         (a  (dotprod (list x1 y1 z1) (list x2 y2 z2)))
         (x2 (- x2 (* a x1)))
         (y2 (- y2 (* a y1)))
         (z2 (- z2 (* a z1)))
         (m2 (magnitude (list x2 y2 z2)))
         (x2 (/ x2 m2))
         (y2 (/ y2 m2))
         (z2 (/ z2 m2)))
    (setf (aref mx 0 0) x1) (setf (aref mx 0 1) x2) (setf (aref mx 0 2) x3)
    (setf (aref mx 1 0) y1) (setf (aref mx 1 1) y2) (setf (aref mx 1 2) y3)
    (setf (aref mx 2 0) z1) (setf (aref mx 2 1) z2) (setf (aref mx 2 2) z3)
    (setf (aref mx 0 3) tx) (setf (aref mx 1 3) ty) (setf (aref mx 2 3) tz)
    mx))


(defun plane-transform ( plane matrix )
; Transformed-Plane = Plane * Matrix
; plane is defined as ((a b c) d). (a b c) is unit normal. d is -(distance).
  (let* ((new-a nil)
         (new-b nil)
         (new-c nil)
         (new-d nil)
         (old-unit-normal (car plane))
         (old-d (cadr plane))
         (old-a (first  old-unit-normal))
         (old-b (second old-unit-normal))
         (old-c (third  old-unit-normal))
         (mag nil))
    (setf new-a (+ (* old-a (aref matrix 0 0)) (* old-b (aref matrix 1 0))
```

```
                    (* old-c (aref matrix 2 0))))
    (setf new-b (+ (* old-a (aref matrix 0 1)) (* old-b (aref matrix 1 1))
                   (* old-c (aref matrix 2 1))))
    (setf new-c (+ (* old-a (aref matrix 0 2)) (* old-b (aref matrix 1 2))
                   (* old-c (aref matrix 2 2))))
    (setf new-d (+ (* old-a (aref matrix 0 3)) (* old-b (aref matrix 1 3))
                   (* old-c (aref matrix 2 3)) old-d))
    (setf mag (magnitude (list new-a new-b new-c)))
    (if (< (abs mag) 0.0000001)
        (print "Error in PlaneTransform")
        (list (list (/ new-a mag) (/ new-b mag) (/ new-c mag))
              (/ new-d mag)))))


(defun plane-distance (plane velocity position)
; Plane (X - Q)N = 0 ,  straight line X = P + tA.
; t = ( Q - P )N / ( AN )  if A is normalized then t is signed distance.
; if t is infinitive then plane-distance  returnes nil.
; plane-distance returns t.
  (let* ((A (normalize-vector velocity))
         (N (first plane))
         (dis (- (second plane)))
         (Q   (magvect dis N)) ; magvect = const * vector
         (P   position)
         (Q_P (vectsub Q P))
         (AN  (dotprod A N))
         (numerator (dotprod Q_P N)))
    (if (< (abs AN) 0.0000001)  ; no crossing
        nil                     ; returns nil
        (/ numerator AN))))


(defun plane-intersection (a-line a-plane)
; a-line  ((direction) (point))  X = P + tA.
; a-plane ((unit-normal) -dist)  (X - Q)N = 0.
  (let* ((velocity (normalize-vector (first a-line)))
         (position (second a-line))
         (t-value (plane-distance a-plane velocity position)))
    (if t-value
        (vectadd position (magvect t-value velocity))
        nil))) ; no intersection


(defun plane-normal-distance (a-plane a-point)
; vector-type-plane (a b c d)
; paul-type-point   transpose(x y z 1)
  (let* ((unit-normal (first  a-plane))
         (dis         (second a-plane))
         (vector-type-plane (reverse (cons dis (reverse unit-normal))))
         (paul-type-point   (reverse (cons 1   (reverse a-point)))))
    (dotprod vector-type-plane paul-type-point)))


(defun rotatemat(axis angle) ; array index starts from 0 not 1.
; return rotatematrix   angle :radian    axis : x y or z
  (let ((mat (ident))
```

```
        (cosa (cos angle))
        (sina (sin angle)))
    (case axis
          (x-axis
            (setf (aref mat 1 1)  cosa)      (setf (aref mat 1 2) (- sina))
            (setf (aref mat 2 1)  sina)      (setf (aref mat 2 2)  cosa))
          (y-axis
            (setf (aref mat 0 0)  cosa)      (setf (aref mat 0 2)  sina)
            (setf (aref mat 2 0) (- sina))  (setf (aref mat 2 2)  cosa))
          (z-axis
            (setf (aref mat 0 0)  cosa)      (setf (aref mat 0 1) (- sina))
            (setf (aref mat 1 0)  sina)  (setf (aref mat 1 1)  cosa)))
    mat))  ; returns this value.


(defun to-body-transform (inv-H points-wrt-earth)
; returns points-wrt-body
  (if (listp (first points-wrt-earth))            ; test multi-points
      (do ((points points-wrt-earth (cdr points)) ; multi-points case
           (out-points nil))
          ((null points) (reverse out-points))
        (setf out-points (cons (transform inv-H (car points)) out-points)))
      (transform inv-H points-wrt-earth)))        ; single point case


(defun to-earth-transform (H points-wrt-body)
; returns points-wrt-earth
  (if (listp (first points-wrt-body))             ; test multi-points
      (do ((points points-wrt-body (cdr points))  ; multi-points case
           (out-points nil))
          ((null points) (reverse out-points))
        (setf out-points (cons (transform H (car points)) out-points)))
      (transform H points-wrt-body)))             ; single point case


(defun transform(mat point)  ; array index starts from 0 not 1.
  (let ((x (car point))
        (y (cadr point))
        (z (if (caddr point) (caddr point) 0)))
    (list (+ (* x (aref mat 0 0)) (* y (aref mat 0 1)) (* z (aref mat 0 2))
             (aref mat 0 3))
          (+ (* x (aref mat 1 0)) (* y (aref mat 1 1)) (* z (aref mat 1 2))
             (aref mat 1 3))
          (+ (* x (aref mat 2 0)) (* y (aref mat 2 1)) (* z (aref mat 2 2))
             (aref mat 2 3)))))


(defun transmat (x y z)
; returns translational marix
  (let ((matrix (ident)))
    (setf (aref matrix 0 3) x)
    (setf (aref matrix 1 3) y)
    (setf (aref matrix 2 3) z)
    matrix))


(defun transpose (mat)
  (let ((matrix (make-array '(4 4))))
```

```
    (dotimes (i 4)
      (dotimes (j 4)
        (setf (aref matrix i j) (aref mat j i))))
    matrix))


(defun unit-crossprod (vect1 vect2)
; generate unitnormal vector of vect1 X vect2
  (let* ((x1 (first  vect1)) (x2 (first  vect2))
         (y1 (second vect1)) (y2 (second vect2))
         (z1 (third  vect1)) (z2 (third  vect2))
         (x (- (* y1 z2) (* y2 z1)))
         (y (- (* x2 z1) (* x1 z2)))
         (z (- (* x1 y2) (* x2 y1)))
         (m (sqrt (+ (* x x) (* y y) (* z z)))))
    (list (/ x m) (/ y m) (/ z m))))


(defun vectadd (vect1 vect2)
; vectsub = vect1 + vect2
; no limit in dimension
  (mapcar '+ vect1 vect2))


(defun vectsub (vect1 vect2)
; vectsub = vect1 - vect2
; no limit in dimension
  (mapcar '- vect1 vect2))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  overlap-foothold-finder definition
;
;*****************************************************************

(defflavor overlap-foothold-finder(adjacent-leg-numbers)
           (foothold-finder)
  :initable-instance-variables)


(defmethod (overlap-foothold-finder :initti)
           (leg-name)
  (cond ((equal leg-name 'leg1)
         (setf adjacent-leg-numbers '(3))
         (setf sixteen-footholds
               '(( 9.0  4.3) ( 9.0  3.3) ( 9.0  2.3) ( 9.0  1.3)
                 ( 8.0  4.3) ( 8.0  3.3) ( 8.0  2.3) ( 8.0  1.3)
                 ( 7.0  4.3) ( 7.0  3.3) ( 7.0  2.3) ( 7.0  1.3)
                 ( 6.0  4.3) ( 6.0  3.3) ( 6.0  2.3) ( 6.0  1.3)
                 ( 5.0  4.3) ( 5.0  3.3) ( 5.0  2.3) ( 5.0  1.3)
                 ( 4.0  4.3) ( 4.0  3.3) ( 4.0  2.3) ( 4.0  1.3)
                 ( 3.0  4.3) ( 3.0  3.3) ( 3.0  2.3) ( 3.0  1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) (  9.5  2.7339 0))
                 ((0 -0.3420 -0.9397) (  9.5  2.7339 0))
                 ((0 -0.3420 -0.9397) (  2.5  2.7339 0))
                 ((0  0.3420 -0.9397) (  2.5  2.7339 0)))))
        ((equal leg-name 'leg2)
         (setf adjacent-leg-numbers '(4))
         (setf sixteen-footholds
               '(( 9.0 -4.3) ( 9.0 -3.3) ( 9.0 -2.3) ( 9.0 -1.3)
                 ( 8.0 -4.3) ( 8.0 -3.3) ( 8.0 -2.3) ( 8.0 -1.3)
                 ( 7.0 -4.3) ( 7.0 -3.3) ( 7.0 -2.3) ( 7.0 -1.3)
                 ( 6.0 -4.3) ( 6.0 -3.3) ( 6.0 -2.3) ( 6.0 -1.3)
                 ( 5.0 -4.3) ( 5.0 -3.3) ( 5.0 -2.3) ( 5.0 -1.3)
                 ( 4.0 -4.3) ( 4.0 -3.3) ( 4.0 -2.3) ( 4.0 -1.3)
                 ( 3.0 -4.3) ( 3.0 -3.3) ( 3.0 -2.3) ( 3.0 -1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) (  9.5 -2.7339 0))
                 ((0 -0.3420 -0.9397) (  9.5 -2.7339 0))
                 ((0 -0.3420 -0.9397) (  2.5 -2.7339 0))
                 ((0  0.3420 -0.9397) (  2.5 -2.7339 0)))))
        ((equal leg-name 'leg3)
         (setf adjacent-leg-numbers '(1 5))
         (setf sixteen-footholds
               '(( 3.0  4.3) ( 3.0  3.3) ( 3.0  2.3) ( 3.0  1.3)
                 ( 2.0  4.3) ( 2.0  3.3) ( 2.0  2.3) ( 2.0  1.3)
                 ( 1.0  4.3) ( 1.0  3.3) ( 1.0  2.3) ( 1.0  1.3)
                 ( 0.0  4.3) ( 0.0  3.3) ( 0.0  2.3) ( 0.0  1.3)
                 (-1.0  4.3) (-1.0  3.3) (-1.0  2.3) (-1.0  1.3)
                 (-2.0  4.3) (-2.0  3.3) (-2.0  2.3) (-2.0  1.3)
                 (-3.0  4.3) (-3.0  3.3) (-3.0  2.3) (-3.0  1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) ( 3.5  2.7339 0))
                 ((0 -0.3420 -0.9397) ( 3.5  2.7339 0))
                 ((0 -0.3420 -0.9397) (-3.5  2.7339 0))
                 ((0  0.3420 -0.9397) (-3.5  2.7339 0)))))
        ((equal leg-name 'leg4)
         (setf adjacent-leg-numbers '(2 6))
         (setf sixteen-footholds
               '(( 3.0  -4.3) ( 3.0  -3.3) ( 3.0  -2.3) ( 3.0  -1.3)
```

```
                ( 2.0  -4.3) ( 2.0  -3.3) ( 2.0  -2.3) ( 2.0  -1.3)
                ( 1.0  -4.3) ( 1.0  -3.3) ( 1.0  -2.3) ( 1.0  -1.3)
                ( 0.0  -4.3) ( 0.0  -3.3) ( 0.0  -2.3) ( 0.0  -1.3)
                (-1.0  -4.3) (-1.0  -3.3) (-1.0  -2.3) (-1.0  -1.3)
                (-2.0  -4.3) (-2.0  -3.3) (-2.0  -2.3) (-2.0  -1.3)
                (-3.0  -4.3) (-3.0  -3.3) (-3.0  -2.3) (-3.0  -1.3)))
        (setf four-lines
              '(((0  0.3420 -0.9397) ( 3.5 -2.7339 0))
                ((0 -0.3420 -0.9397) ( 3.5 -2.7339 0))
                ((0 -0.3420 -0.9397) (-3.5 -2.7339 0))
                ((0  0.3420 -0.9397) (-3.5 -2.7339 0)))))
       ((equal leg-name 'leg5)
        (setf adjacent-leg-numbers '(3))
        (setf sixteen-footholds
              '((-3.0  4.3) (-3.0  3.3) (-3.0  2.3) (-3.0  1.3)
                (-4.0  4.3) (-4.0  3.3) (-4.0  2.3) (-4.0  1.3)
                (-5.0  4.3) (-5.0  3.3) (-5.0  2.3) (-5.0  1.3)
                (-6.0  4.3) (-6.0  3.3) (-6.0  2.3) (-6.0  1.3)
                (-7.0  4.3) (-7.0  3.3) (-7.0  2.3) (-7.0  1.3)
                (-8.0  4.3) (-8.0  3.3) (-8.0  2.3) (-8.0  1.3)
                (-9.0  4.3) (-9.0  3.3) (-9.0  2.3) (-9.0  1.3)))
        (setf four-lines
              '(((0  0.3420 -0.9397) (-2.5  2.7339 0))
                ((0 -0.3420 -0.9397) (-2.5  2.7339 0))
                ((0 -0.3420 -0.9397) (-9.5  2.7339 0))
                ((0  0.3420 -0.9397) (-9.5  2.7339 0)))))
       ((equal leg-name 'leg6)
        (setf adjacent-leg-numbers '(4))
        (setf sixteen-footholds
              '((-3.0 -4.3) (-3.0 -3.3) (-3.0 -2.3) (-3.0 -1.3)
                (-4.0 -4.3) (-4.0 -3.3) (-4.0 -2.3) (-4.0 -1.3)
                (-5.0 -4.3) (-5.0 -3.3) (-5.0 -2.3) (-5.0 -1.3)
                (-6.0 -4.3) (-6.0 -3.3) (-6.0 -2.3) (-6.0 -1.3)
                (-7.0 -4.3) (-7.0 -3.3) (-7.0 -2.3) (-7.0 -1.3)
                (-8.0 -4.3) (-8.0 -3.3) (-8.0 -2.3) (-8.0 -1.3)
                (-9.0 -4.3) (-9.0 -3.3) (-9.0 -2.3) (-9.0 -1.3)))
        (setf four-lines
              '(((0  0.3420 -0.9397) (-2.5 -2.7339 0))
                ((0 -0.3420 -0.9397) (-2.5 -2.7339 0))
                ((0 -0.3420 -0.9397) (-9.5 -2.7339 0))
                ((0  0.3420 -0.9397) (-9.5 -2.7339 0)))))

       )
  (setf tkm-calculator (send owner :tkm-calculator))
  )



(defmethod (overlap-foothold-finder :get-possible-footholds)
           (estimated-footholds H inv-H)
; returns possible-footholds wrt body
; find-possible-footholds function tests obstacles
  (to-body-transform
    inv-H
    (send self :get-rid-of-overlap
          (send self :find-possible-footholds
                (to-earth-transform H estimated-footholds)))))


(defmethod (overlap-foothold-finder :get-rid-of-overlap)
           (footholds-wrt-earth)
```

```
  (let* ((ajacent-legs
           (mapcar
             #'(lambda (leg-num)
                 (send owner :nth-leg leg-num))
             adjacent-leg-numbers))
         (adjacent-legs-in-possible-interaction
           (remove
             nil
             (mapcar #'(lambda (leg)
                         (if (send leg :place-able) ;no interaction
                             nil
                             leg))
                     ajacent-legs))))
    (send self :remove-overlapped-foothold
          footholds-wrt-earth
          adjacent-legs-in-possible-interaction)))


(defmethod (overlap-foothold-finder :remove-overlapped-foothold
           (footholds-wrt-earth legs-in-possible-interaction)
  (do ((legs legs-in-possible-interaction (cdr legs))
       (out-footholds footholds-wrt-earth)
       (overlap-foothold))
      ((null legs) out-footholds)
    (setf overlap-foothold (send (car legs) :foothold))
    (setf out-footholds
          (remove overlap-foothold
                  out-footholds
                  :test #'(lambda (x1 x2)
                            (send self :overlap-p x1 x2))
                  ))))


(defmethod (overlap-foothold-finder :overlap-p)
           (x1 x2)
  (let ((x1-integer (mapcar #'truncate x1))
        (x2-integer (mapcar #'truncate x2)))
    (equal x1-integer x2-integer)))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  overlap-leg definition
;
;*****************************************************************


(defflavor overlap-leg ()
           (leg)
  )



(defmethod (overlap-leg :initti)
           (H)
    (setf contact-sensor  (make-instance 'contact-sensor  :owner self))
    (setf executor        (make-instance 'executor        :owner self))
    (setf control-machine (make-instance 'control-state-machine :owner self))
    (setf plan-machine    (make-instance 'plan-state-machine    :owner self))
    (setf tkm-calculator  (make-instance 'overlap-tkm-calculator  :owner self))
    (setf foothold-finder (make-instance 'overlap-foothold-finder :owner self))
    (setf foothold (send executor :initti name H))
    (send contact-sensor :initti name)
    (send control-machine :initti name)
    (send plan-machine :initti name)
    (send tkm-calculator :initti name)
    (send foothold-finder :initti name))



(defmethod (overlap-leg :nth-leg)
           (leg-num)
  (send owner :nth-leg leg-num))


(defmethod (overlap-leg :foothold)
           ()
  foothold)
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  overlap-robot definition
;
;*****************************************************************


(defflavor overlap-robot()
           (robot)
  :initable-instance-variables
  :gettable-instance-variables)



(defmethod (overlap-robot :initti)
           ()
  (send graph-asv :init-data)
  (setf vision-system (make-instance 'vision-system :owner self))
  (send vision-system :initti)
  (setf joystick (make-instance 'joystick))
  (send joystick :reset)
  (empty-queue lift-queue)
  (setf lift-flag t)
  (let ((H))
    (setf body (make-instance 'body :owner self))
    (setf H (send body :initti))
    (setf legs (list
                 (make-instance 'overlap-leg :name 'leg1  :owner self)
                 (make-instance 'overlap-leg :name 'leg2  :owner self)
                 (make-instance 'overlap-leg :name 'leg3  :owner self)
                 (make-instance 'overlap-leg :name 'leg4  :owner self)
                 (make-instance 'overlap-leg :name 'leg5  :owner self)
                 (make-instance 'overlap-leg :name 'leg6  :owner self)
                 ))
    (mapcar #'(lambda (a-leg) (send a-leg :initti H)) legs))
  )



(defmethod (overlap-robot :nth-leg)
           (leg-num)
; nth starts counting from zero.
; leg-num starts from one.
  (nth (- leg-num 1) legs))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
; overlap-tkm-calculator definition
;
;*****************************************************************


(defflavor overlap-tkm-calculator()
          (tkm-calculator)
  :initable-instance-variables)


(defmethod (overlap-tkm-calculator :initti)
           (leg-name)
  (cond ((equal leg-name 'leg1)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -9.5) ((0 0.9397  0.3420) -2.569))
                 (((0 0 1) 5.7313) ((1 0 0) -2.5) ((0 0.9397 -0.3420) -2.569)))))
        ((equal leg-name 'leg2)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -9.5) ((0 0.9397  0.3420)  2.569))
                 (((0 0 1) 5.7313) ((1 0 0) -2.5) ((0 0.9397 -0.3420)  2.569)))))
        ((equal leg-name 'leg3)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -3.5) ((0 0.9397  0.3420) -2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  3.5) ((0 0.9397 -0.3420) -2.569)))))
        ((equal leg-name 'leg4)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -3.5) ((0 0.9397  0.3420)  2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  3.5) ((0 0.9397 -0.3420)  2.569)))))
        ((equal leg-name 'leg5)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0)  2.5) ((0 0.9397  0.3420) -2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  9.5) ((0 0.9397 -0.3420) -2.569)))))
        ((equal leg-name 'leg6)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0)  2.5) ((0 0.9397  0.3420)  2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  9.5) ((0 0.9397 -0.3420)  2.569)))))
        )
  )
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************
;
;  plan-state flavor definition
;
;*****************************************************

(defflavor plan-state ((decision nil) (observation nil) (command nil)
                       (condition nil))
           (state)
  :initable-instance-variables)


(defmethod (plan-state :generate-command)
           ()
  command)


(defmethod (plan-state :change)
           (given-decision observed-state given-condition)
  (cond ((and decision (listp decision))
         (cond ((equal given-decision (first decision))
                (first next-state))
               ((equal given-decision (second decision))
                (second next-state))
               (t self)))
        (condition
         (if (and (equal given-condition condition)
                  (equal observed-state observation))
             next-state
             self))
        (t
         (cond ((equal observed-state observation)
                next-state)
               ((equal given-decision decision)
                next-state)
               (t self)))))


;*****************************************************
;
;  plan-state-machinie flavor definition
;
;*****************************************************

(defflavor plan-state-machine ((decision nil) (observation nil)
                               (condition nil) (lift-ready-flag nil)
                               control-machine)
           (state-machine)
  :initable-instance-variables)


(defmethod (plan-state-machine :initti)
           (leg-name)
  (if (member leg-name '(leg1 leg4 leg5))
```

```
      (send self :init-plan-machine 'eligible-to-lift)
      (send self :init-plan-machine 'available-leg))
  (setf control-machine (send owner :control-machine)))


(defmethod (plan-state-machine :init-plan-machine)
           (a-state-name)
  (let (available-leg planned-contact eligible-to-lift
        planned-lift  actual-lift planned-exchange)
    (setf actual-lift
          (make-instance 'plan-state
                         :name 'actual-lift
                         :observation 'ready
                         :command 'recover-command))
    (setf planned-lift
          (make-instance 'plan-state
                         :name 'planned-lift :condition 'stable-without
                         :observation 'support
                         :next-state actual-lift))
    (setf planned-exchange
          (make-instance 'plan-state
                         :name 'planned-exchange :condition 'interlock-confirm
                         :observation 'support
                         :next-state actual-lift))
    (setf eligible-to-lift
          (make-instance 'plan-state
                         :name 'eligible-to-lift
                         :decision '(lift exchange)
                         :next-state (list planned-lift planned-exchange)))
    (setf planned-contact
          (make-instance 'plan-state
                         :name 'planned-contact :observation 'contact
                         :command 'deploy-command
                         :next-state eligible-to-lift))
    (setf available-leg
          (make-instance 'plan-state
                         :name 'available-leg :decision 'place
                         :next-state planned-contact))
    (send actual-lift :set-next-state available-leg)

    (setf state (cond ((equal a-state-name (send available-leg :state-name))
                       available-leg)
                      ((equal a-state-name (send planned-contact :state-name))
                       planned-contact)
                      ((equal a-state-name (send eligible-to-lift :state-name))
                       eligible-to-lift)
                      ((equal a-state-name (send planned-lift :state-name))
                       planned-lift)
                      ((equal a-state-name (send planned-exchange :state-name))
                       planned-exchange)
                      ((equal a-state-name (send actual-lift :state-name))
                       actual-lift)))
    )
  )


;(defmethod (plan-state-machine :change :before)
;          ()
;  (setf observation (send control-machine :state-name))
;  (cond ((and (equal (send state :state-name) 'planned-exchange)
;              (send owner :interlock-confirm)
;              (send owner :stable-without-p)
```

```
;               (send owner :lift-ok))
;          (setf lift-ready-flag t)
;          (setf condition 'interlock-confirm))
;         ((and (equal (send state :state-name) 'planned-lift)
;               (send owner :stable-without-p)
;               (send owner :lift-ok))
;          (setf lift-ready-flag t)
;          (setf condition 'stable-without))
;         (t
;          (setf condition nil)
;          (setf lift-ready-flag nil)))
;  )


(defmethod (plan-state-machine :change)
           ()
  (setf observation (send control-machine :state-name))
  (cond ((and (equal (send state :state-name) 'planned-exchange)
              (send owner :interlock-confirm)
              (send owner :stable-without-p)
              (send owner :lift-ok))
         (setf lift-ready-flag t)
         (setf condition 'interlock-confirm))
        ((and (equal (send state :state-name) 'planned-lift)
              (send owner :stable-without-p)
              (send owner :lift-ok))
         (setf lift-ready-flag t)
         (setf condition 'stable-without))
        (t
         (setf condition nil)
         (setf lift-ready-flag nil)))
  (setf state (send state :change decision observation condition))
;  )

;(defmethod (plan-state-machine :change :after)
;          ()
  (send control-machine :send-command
                (send state :generate-command))
  (if (and lift-ready-flag
           (equal (send self :state-name) 'actual-lift))
      (send owner :lifted)))


(defmethod (plan-state-machine :send-decision)
           (a-decision)
  (setf decision a-decision))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-


;*****************************************************
;
;  robot flavor definition
;
;*****************************************************

(defflavor robot(legs body vision-system joystick
                 (lift-able-legs nil)
                 (place-able-legs nil) (supporting-legs nil)
                 (supporting-p-legs nil)
                 (joy-command '(0 0 0)) lift-queue lift-flag)
           ()
  :initable-instance-variables
  :gettable-instance-variables)


(defmethod (robot :initti)
           ()
  (send graph-asv :init-data)
  (setf vision-system (make-instance 'vision-system :owner self))
  (send vision-system :initti)
  (setf joystick (make-instance 'joystick))
  (send joystick :reset)
  (empty-queue lift-queue)
  (setf lift-flag t)
  (let ((H))
    (setf body (make-instance 'body :owner self))
    (setf H (send body :initti))
    (setf legs (list
                 (make-instance 'leg :name 'leg1  :owner self)
                 (make-instance 'leg :name 'leg2  :owner self)
                 (make-instance 'leg :name 'leg3  :owner self)
                 (make-instance 'leg :name 'leg4  :owner self)
                 (make-instance 'leg :name 'leg5  :owner self)
                 (make-instance 'leg :name 'leg6  :owner self)
                 ))
    (mapcar #'(lambda (a-leg) (send a-leg :initti H)) legs))
  )


(defmethod (robot :find-lift-able-legs)
           ()
  (delete nil (mapcar #'(lambda (a-leg) (send a-leg :lift-able)) legs)))


(defmethod (robot :find-place-able-legs)
           ()
  (delete nil (mapcar #'(lambda (a-leg) (send a-leg :place-able)) legs)))


(defmethod (robot :find-supporting-legs)
           ()
  (delete nil (mapcar #'(lambda (a-leg) (send a-leg :supporting)) legs)))
```

```
(defmethod (robot :find-supporting-p-legs)
           ()
  (delete nil (mapcar #'(lambda (a-leg) (send a-leg :supporting-p)) legs)))


(defmethod (robot :get-body-rotate-rate1)
           ()
  (send body :get-body-rotate-rate1))


(defmethod (robot :get-body-rotate-rate10)
           ()
  (send body  :get-body-rotate-rate10))


(defmethod (robot :get-body-trans-rate1)
           ()
  (send body :get-body-trans-rate1))


(defmethod (robot :get-body-trans-rate10)
           ()
  (send body :get-body-trans-rate10))


(defmethod (robot :get-estimated-support-plane)
           ()
  (send body :get-estimated-support-plane))


(defmethod (robot :get-H1)
           ()
  (send body :get-H1))


(defmethod (robot :get-H6)
           ()
  (send body :get-H6))


(defmethod (robot :get-H10)
           ()
  (send body :get-H10))


(defmethod (robot :get-inv-H1)
           ()
  (send body :get-inv-H1))


(defmethod (robot :get-inv-H6)
           ()
  (send body :get-inv-H6))
```

```
(defmethod (robot :get-inv-H10)
           ()
  (send body :get-inv-H10))


(defmethod (robot :lift-ok)
           (leg-name)
  (cond (lift-flag
         (cond ((equal leg-name (send (first lift-queue) :name))
                (setf lift-flag nil)
                t)
               (t
                nil)))
        (t nil)))


(defmethod (robot :lifted)
           (leg-name)
  (if (equal leg-name (send (first lift-queue) :name))
      (dequeue lift-queue)
      (print (list "error in lifting" leg-name))))


(defmethod (robot :permitted-cell)
           (t-cell)
  (send vision-system :permitted-cell t-cell))


(defmethod (robot :scanning)
           ()
  (send vision-system :scanning))


(defmethod (robot :stable-without-p)
           (a-leg)
  (send body :stable-p
            (remove a-leg supporting-p-legs)))


(defmethod (robot :terrain-point)
           (t-cell)
  (send vision-system :terrain-point t-cell))



;*************************************************************
;
; prolog interface robot methods
;
;*************************************************************
```

```
(defmethod (robot :at-tkm-limit)
           ()
  (let ((limit-leg
         (car (delete nil
              (mapcar #'(lambda (a-leg) (send a-leg :TKM-limit)) lift-able-legs)))))
    (setf supporting-legs (remove limit-leg
                                  supporting-legs))
    (setf lift-able-legs (remove limit-leg
                                 lift-able-legs))
    limit-leg))



(defmethod (robot :check-stability-p)
           ()
  (send body :stable-p-m supporting-p-legs (first lift-queue)))


(defmethod (robot :check-tkm-limit-p)
           ()
  (delete nil
          (mapcar #'(lambda (a-leg) (send a-leg :TKM-limit-p)) supporting-p-legs)))


(defmethod (robot :do-recovery)
           ()
  (car
    (delete nil
            (mapcar #'(lambda (a-leg) (send a-leg :with-foothold)) place-able-legs))))


(defmethod (robot :execute-planned-motion)
           ()
  (mapcar #'(lambda (a-leg) (send a-leg :do-planned-motion)) legs))


(defmethod (robot :graphical-display)
           ()
  (send graph-asv :display (send body :get-H1)
           (mapcar #'(lambda (a-leg) (send a-leg :leg-pos-wrt-body)) legs))
  )


(defmethod (robot :has-more-tkm)
           (leg1 leg2)
  (> (send leg1 :tkm)
     (send leg2 :tkm)))


(defmethod (robot :leg-with-new-foothold)
           ()
; return a-leg with new-foothold.
  (do ((new-foothold-flags (mapcar #'(lambda (a-leg) (send a-leg :new-foothold)) place-a
e-legs)
                           (mapcar #'(lambda (a-leg) (send a-leg :new-foothold)) place-a
e-legs))
       (a-leg nil))
```

```
      ((or (nil-list new-foothold-flags)
           a-leg)
       (if a-leg a-leg nil))
    (setf a-leg (send self :max-sm-leg nil))))


(defmethod (robot :max-sm-leg)
           (a-leg)
; max-sm-leg without supporting a-leg
  (let (legs-with-foothold)
    (cond (place-able-legs
           (setf legs-with-foothold
                 (remove nil (mapcar #'(lambda (leg)
                                         (if (send leg :has-foothold-p)
                                             leg
                                             nil))
                                     place-able-legs)))
           (cond (legs-with-foothold
                  (do ((legs (cdr legs-with-foothold) (cdr legs))
                       (largest-leg (car legs-with-foothold) largest-leg)
                       (temp-support-legs (remove a-leg supporting-legs)))
                      ((null legs)
                       (if (send body :stable (cons largest-leg temp-support-legs))
                           largest-leg
                           nil))
                    (if (send body :more-stable temp-support-legs
                               (car legs) largest-leg)
                        (setf largest-leg  (car legs)))))
                 (t nil)))
          (t nil))))


(defmethod (robot :modify-command)
           ()
  (send body :modify-command))


(defmethod (robot :wait-for-lift)
           ()
  (delete nil
          (mapcar #'(lambda (a-leg) (send a-leg :lift-not-done)) supporting-p-legs)))


(defmethod (robot :read-joystick)
           ()
  (let ((joy-value (send joystick :get-joy-value)))
    (setf joy-command
          (reverse (cdr (reverse (send joystick :get-joy-value)))))
    (if (fourth joy-value)
        nil
        t)))


(defmethod (robot :restore-command)
           ()
  (send body :restore-command))


(defmethod (robot :before :send-decision)
```

```
            (leg1 leg2 a-decision)
  (cond ((equal a-decision 'exchange)
         (enqueue lift-queue leg1))
        ((equal a-decision 'lift)
         (enqueue lift-queue leg1))))


(defmethod (robot :send-decision)
            (leg1 leg2 a-decision)
  (cond ((equal a-decision 'exchange)
         (send leg1 :send-decision a-decision)
         (send leg2 :send-decision 'place)
         (send leg1 :send-exchange leg2))
        (t
         (send leg1 :send-decision a-decision))))


(defmethod (robot :smallest-tkm-leg)
            ()
; select smallest-TKM-leg
; tkm is nil or positive
  (do ((legs (cdr lift-able-legs) (cdr legs))
       (smallest-leg  (car lift-able-legs))
       (smallest-tkm nil) (tkm nil))
      ((null legs) smallest-leg)
    (setf smallest-tkm (if (send smallest-leg :tkm)
                           (send smallest-leg :tkm) -1000))
    (setf tkm (if (send (car legs) :tkm)
                  (send (car legs) :tkm) -1000))
    (if (> smallest-tkm tkm) (setf smallest-leg (car legs)))
    (if (and (equal smallest-tkm -1000) (equal tkm -1000))
        "Error : more than one legs are out of kinematic limit")))


(defmethod (robot :slow-down-robot)
            ()
  (send body :slow-down))


(defmethod (robot :speed-up-robot)
            ()
  (send body :speed-up))


(defmethod (robot :stable)
            ()
  (send body :stable supporting-legs))


(defmethod (robot :stable_m)
            ()
  (send body :stable-m supporting-legs))


(defmethod (robot :stable-without)
            (a-leg)
  (send body :stable (remove a-leg supporting-legs)))
```

```
(defmethod (robot :update-robot-status)
           ()
  (setf lift-flag t)
  (setf lift-able-legs (send self :find-lift-able-legs))
  (setf place-able-legs (send self :find-place-able-legs))
  (setf supporting-legs (send self :find-supporting-legs))
  (setf supporting-p-legs (send self :find-supporting-p-legs))
  (mapcar #'(lambda (a-leg) (send a-leg :update-tkm-p)) supporting-p-legs)
  (if (send self :check-tkm-limit-p)
      (send body :stop-motion (send self :check-tkm-limit-p))
      (send body :restore-motion))
  (if (not (send self :check-stability-p))
      (send body :modify-command-p)
      (send body :restore-command-p))
  (send body :calculate-motion joy-command legs)
  (mapcar #'(lambda (a-leg) (send a-leg :select-foothold)) place-able-legs)
  (mapcar #'(lambda (a-leg) (send a-leg :update-tkm)) supporting-legs))




(defun create-terrain ()
  (send graph-terrain :create))


(defun kill-terrain()
  (send graph-terrain :kill))



;*******************************************************
;
;  prolog interface functions
;
;*******************************************************


(defun at_tkm_limit()
  (send asv :at-tkm-limit))



(defun do_recovery()
  (send asv :do-recovery))



(defun execute_planned_motion()
  (send asv :execute-planned-motion))



(defun graphical_display()
  (send asv :graphical-display))


(defun has_more_tkm(leg1 leg2)
  (send asv :has-more-tkm leg1 leg2))
```

```
(defun inits()
  (send asv :initti))


(defun leg_with_new_foothold()
  (send asv :leg-with-new-foothold))


(defun max_sm_leg(a-leg)
  (send asv :max-sm-leg a-leg))


(defun modify_command()
  (send asv :modify-command))


(defun read_joystick()
  (send asv :read-joystick))


(defun restore_command()
  (send asv :restore-command))


(defun send_decision(leg1 leg2 a-decision)
  (send asv :send-decision leg1 leg2 a-decision))


(defun smallest_tkm_leg()
  (send asv :smallest-tkm-leg))


(defun slow_down_robot()
  (send asv :slow-down-robot))


(defun speed_up_robot()
  (send asv :speed-up-robot))


(defun stable_p()
  (send asv :stable))


(defun stable_p_m()
  (send asv :stable_m))


(defun stable_without(a-leg)
  (send asv :stable-without a-leg))
```

```
(defun update_robot_status()
  (send asv :update-robot-status))
```

```
(fs:add-logical-pathname-host "robot" "root5"
                              '(("kwak" "supermac:kwak:")))
```

```
;;; -*- Mode:Common-Lisp; Package:USER; Base:10 -*-
;*****************************************************************
;
;  top level motion planning coordinator
;
;*****************************************************************



(defun my-monitor (&rest args)
  (let ((x (mapcar #'my-output args)))
    (if (remove nil x)
        (my-print x)))
  t)

(defun my-output(arg)
  (cond ((typep arg 'leg) (send arg :name))
        ((typep arg 'atom) arg)
        ((typep arg 'list) (cons (my-output (car arg))
                                 (my-output (cdr arg))))
        (t 'error)))




;
;

(defmacro retract(predicate &optional (argument t))
  `(cond ((not (boundp (quote ,predicate)))
          nil)
         ((and ,predicate (equal ,argument '?))
          (setf ,predicate (cdr ,predicate))
          t)
         ((member ,argument ,predicate :test 'equal)
          (setf ,predicate (remove ,argument ,predicate :count 1))
          t)
         (t nil)))


(defmacro assorta(predicate &optional (argument t))
  `(cond ((not (boundp (quote ,predicate)))
          (setf ,predicate nil)
          (setf ,predicate (cons ,argument ,predicate)))
         (t (setf ,predicate (cons ,argument ,predicate)))))


(defmacro assertz(predicate &optional (argument t))
  `(cond ((not (boundp (quote ,predicate)))
          (setf ,predicate nil)
          (setf ,predicate (append ,predicate (list ,argument))))
         (t (setf ,predicate (append ,predicate (list ,argument))))))


(defmacro match(predicate &optional (argument t))
  `(cond ((not (boundp (quote ,predicate)))
          nil)
         ((member ,argument ,predicate :test 'equal
          t)
         (t nil)))


(defmacro unify(predicate argument)
  `(cond ((not (boundp (quote ,predicate)))
```

```
          nil)
         (t (setf ,argument (car ,predicate)))))



; robot :- initialize, repeat, my_loop, fail.

; initialize :- inits, init_ditch_plan.

; init_dicth_plan :- retract(plan_cycle(_)), retract(plan_state(_)), fail.
; init_ditch_plan :- asserta(plan_cycle(1)), asserta(plan_state(place_legs_in_the_air)).

; my_loop :- get_command, plan, execute, !.

; get_command :- X is read_joystick.


; plan :- ditch_mode, ditch_plan.
; plan :- normal_plan.

; ditch_mode :- ditch_mode(in).   ;; cleared by ditch_plan.
; ditch_mode :- X is at_ditch_area, X == t, asserta(ditch_mode(in)).


; execute :- execute_motion, draw_robot, !.

; execute_motion :- X is execute_planned_motion.

; draw_robot :- X is graphical_display.




(defun robot()
  (create-terrain)
  (robot1)
  (kill-terrain))


(defun robot1 ()
  (initialize)
  (do ()
      ((not (my_loop)))))



(defun initialize()
  (cond ((and (inits)
              (init_ditch_plan))
         t)
        (t nil)))


(defun init_ditch_plan()
  (cond ((and (not (setf ditch_mode nil))
              (not (setf plan_cycle nil))
              (not (setf plan_state nil))
              (not (setf limit_leg nil))
```

```
                (not (setf reduce_speed nil))
                (not (setf front_legs nil))
                (not (setf middle_legs nil))
                (not (setf rear_legs nil))
                (not (setf decision nil))
                (asserta plan_cycle 1)
                (asserta plan_state 'place_legs_in_the_air))
          t)
         (t nil)))



(defun my_loop()
  (process-allow-schedule)
  (cond ((and (get_command)
              (plan)
              (execute))
         t)
         (t nil)))


(defun get_command ()
  (cond (t (read_joystick))
        (t nil)))



(defun plan ()
  (cond ((and (ditch_mode)
              (ditch_plan))
         t)
         ((normal_plan)
          t)
         (t nil)))


(defun ditch_mode ()
  (cond ((match ditch_mode 'in)
         t)
         ((and (at_ditch_area)
               (asserta ditch_mode 'in))
          t)
         (t nil)))


(defun execute ()
  (cond ((and (execute_motion)
              (draw_robot))
         t)
         (t nil)))

(defun execute_motion ()
  (cond (t (execute_planned_motion) t)
        (t nil)))


(defun draw_robot()
  (cond (t (graphical_display) t)
        (t nil)))
```

```
;;*********************************************************
;;
;;    Normal Plan
;;
;;*********************************************************


; normal_plan :- update_robot_state, check_tkm_limit,
;       leg_plan, body_plan, generate_decision, !.

; update_robot_state :- X is update_robot_status.

; check_tkm_limit :- A_leg is at_tkm_limit, A_leg \== nil,
;                    asserta(limit_leg(A_leg,lift)).
; check_tkm_limit.

; leg_plan :- lift_a_leg.
; leg_plan :- exchange_legs.
; leg_plan :- stable.
; leg_plan :- place_a_leg.
; leg_plan :- wait_for_legs.

; stable :- Condition is stable_p, Condition == t.

; lift_a_leg :- stable, A_leg is smallest_tkm_leg, A_leg \== nil,
;               Condition is stable_without(A_leg), Condition == t,
;               asserta(decision(A_leg,_,lift)).

; exchange_legs :- stable, LegA is smallest_tkm_leg, LegA \== nil,
;                  LegB is max_sm_leg(LegA), LegB \== nil,
;                  Condition is has_more_tkm(LegB,LegA),
;                  Condition == t,
;                  asserta(decision(LegA,LegB,exchange)).

; place_a_leg :- A_leg is max_sm_leg(_), A_leg \== nil,
;                asserta(decision(A_leg,_,place)).

; wait_for_legs :- try_new_foothold.
; wait_for_legs :- recovery, asserta(reduce_speed).
; wait_for_legs :- asserta(reduce_speed), restore_limit_leg.

; try_new_foothold :- A_leg is leg_with_new_foothold, A_leg \== nil,
;                     asserta(decision(A_leg,_,place)).

; recovery :- A_leg is do_recovery, A_leg \== nil,
;             asserta(decision(A_leg,_,place)), restore_limit_leg.

; restore_limit_leg :- retract(limit_leg(A_leg,lift)).
; restore_limit_leg.



(defun normal_plan ()
  (cond ((and (update_robot_state)
              (check_tkm_limit)
```

```
                (leg_plan)
                (body_plan)
                (my-monitor limit_leg decision reduce_speed)
                (generate_decision))
          t)
         (t nil)))



(defun update_robot_state()
  (cond ((update_robot_status)
         t)
        (t nil)))


(defun check_tkm_limit()
  (let ((leg))
    (cond ((setf leg (at_tkm_limit))
           (asserta limit_leg (list 'lift leg))
           t)
          (t t)
          (t nil))))




(defun leg_plan()
; OR tree becomes regular "cond" statement.
  (cond ((lift_a_leg)  t)
        ((exchange_leg)  t)
        ((stable)  t)
        ((place_a_leg)  t)
        ((wait_for_legs) t)
        (t nil)))


(defun stable()
  (cond ((stable_p) t)
        (t nil)))




(defun lift_a_leg()
  (let ((leg))
    (cond ((and (stable)
                (setf leg (smallest_tkm_leg))
                (stable_without leg)
                (asserta decision (list 'lift leg)))
           t)
          (t nil))))


(defun exchange_leg()
  (let ((lega) (legb))
    (cond ((and (stable)
                (setf lega (smallest_tkm_leg))
                (setf legb (max_sm_leg lega))
                (has_more_tkm legb lega)
                (asserta decision (list 'exchange lega legb)))
           t)
```

```
          (t nil))))


(defun place_a_leg()
  (let ((leg))
    (cond ((and (setf leg (max_sm_leg nil))
                (asserta decision (list 'place leg)))
           t)
          (t nil))))


(defun wait_for_legs()
; OR
  (cond ((try_new_footholds)  t)
        ((recovery) (asserta reduce_speed)  t)
        ((asserta reduce_speed) (restore_limit_leg)  t)
        (t  nil)))


(defun try_new_footholds()
  (let ((leg))
    (cond ((and (setf leg (leg_with_new_foothold))
                (asserta decision (list 'place leg)))
           t)
          (t nil))))

(defun recovery()
  (let ((leg))
    (cond ((setf leg (do_recovery))
           (asserta decision (list 'place leg))
           (restore_limit_leg)
           t)
          (t nil))))


(defun restore_limit_leg()
; OR
  (cond ((and (unify limit_leg leg)
              (retract limit_leg leg))
         t)
        (t t)
        (t nil)))




;;*************************************************************
;;
;;    Ditch Plan
;;
;;*************************************************************



; ditch_plan :- ditch_plan_done, retract(ditch_mode(in)), idle_cycle.
; ditch_plan :- cycle_planner.



;
;**********  Cycle planner ************
;
```

```
; ditch_plan_done :- plan_cycle(6), retract(plan_cycle(6)), asserta(plan_cycle(1)),
;                    prepare_next_ditch_plan.

; prepare_next_ditch_plan :- move.

; cycle_planner :- one_cycle_done, plan_cycle(N), N1 is N+1, retract(plan_cycle(N)), ass
ta(plan_cycle(N1)),
;                  idle_cycle.
; cycle_planner :- plan_cycle.




;
;**********  Plan Cycle dispatcher ************
;

; one_cycle_done :- plan_state(one_plan_cycle_done), retract(plan_state(one_plan_cycle_d
e)),
;                   initialize_plan_state.


; plan_cycle :- plan_cycle(1), update_robot_state, ditch_plan_cycle_1, body_plan, genera
_decision,!.
; plan_cycle :- plan_cycle(2), update_robot_state, ditch_plan_cycle_2, body_plan, genera
_decision,!.
; plan_cycle :- plan_cycle(3), update_robot_state, ditch_plan_cycle_3, body_plan, genera
_decision,!.
; plan_cycle :- plan_cycle(4), update_robot_state, ditch_plan_cycle_4, body_plan, genera
_decision,!.
; plan_cycle :- plan_cycle(5), update_robot_state, ditch_plan_cycle_5, body_plan, genera
_decision,!.

; idle_cycle :- update_robot_state, body_plan, generate_decision, !.




(defun ditch_plan ()
  (cond ((and (ditch_plan_done)
              (retract ditch_mode 'in)
              (idle_cycle))
         t)
        ((cycle_planner)
         t)
        (t nil)))


;
;**********  Cycle planner ************
;


(defun ditch_plan_done ()
  (cond ((and (match plan_cycle 6)
              (retract plan_cycle 6)
              (asserta plan_cycle 1)
              (prepare_next_ditch_plan))
         t)
        (t nil)))


(defun prepare_next_ditch_plan ()
```

```
  (cond ((move)
         t)
        (t nil)))


(defun cycle_planner ()
  (cond ((and (one_cycle_done)
              (unify plan_cycle N)
              (retract plan_cycle N)
              (assert plan_cycle (+ N 1))
              (idle_cycle))
         t)
        ((plan_cycle)
         t)
        (t nil)))




;
;**********  Plan Cycle Dispatcher *************
;


(defun one_cycle_done ()
  (cond ((and (match plan_state 'one_plan_cycle_done)
              (retract plan_state 'one_plan_cycle_done)
              (initialize_plan_state))
         t)
        (t nil)))



(defun plan_cycle ()
  (cond ((and (match plan_cycle 1)
              (update_robot_state)
              (ditch_plan_cycle_1)
              (body_plan)
              (my-monitor plan_cycle plan_state decision reduce_speed)
              (generate_decision))
         t)
        ((and (match plan_cycle 2)
              (update_robot_state)
              (ditch_plan_cycle_2)
              (body_plan)
              (my-monitor plan_cycle plan_state decision reduce_speed)
              (generate_decision))
         t)
        ((and (match plan_cycle 3)
              (update_robot_state)
              (ditch_plan_cycle_3)
              (body_plan)
              (my-monitor plan_cycle plan_state decision reduce_speed)
              (generate_decision))
         t)
        ((and (match plan_cycle 4)
              (update_robot_state)
              (ditch_plan_cycle_4)
              (body_plan)
              (my-monitor plan_cycle plan_state decision reduce_speed)
              (generate_decision))
         t)
        ((and (match plan_cycle 5)
              (update_robot_state)
```

```
              (ditch_plan_cycle_5)
              (body_plan)
              (my-monitor plan_cycle plan_state decision reduce_speed)
              (generate_decision))
         t)
        (t nil)))


(defun idle_cycle ()
  (cond ((and (update_robot_state)
              (body_plan)
              (my-monitor plan_cycle plan_state decision reduce_speed)
              (generate_decision))
         t)
        (t nil)))




;
;**********  cycles  *********
;


; initialize_plan_state :- asserta(plan_state(start)).


; ditch_plan_cycle_1 :- plan_state(start), retract(plan_state(start)), asserta(plan_stat
place_legs_in_the_air)),
;                       place_legs_in_the_air(back_middle_legs).
; ditch_plan_cycle_1 :- place_legs_in_the_air(back_middle_legs).
; ditch_plan_cycle_1 :- back_middle_legs(forward_rear_legs).
; ditch_plan_cycle_1 :- forward_rear_legs(forward_middle_legs).
; ditch_plan_cycle_1 :- forward_middle_legs(forward_front_legs).
; ditch_plan_cycle_1 :- forward_front_legs(lift_middle_legs_and_move).
; ditch_plan_cycle_1 :- lift_middle_legs_and_move(one_plan_cycle_done).


; ditch_plan_cycle_2 :- plan_state(start), retract(plan_state(start)), asserta(plan_stat
back_middle_legs)),
;                       back_middle_legs(forward_rear_legs).
; ditch_plan_cycle_2 :- back_middle_legs(forward_rear_legs).
; ditch_plan_cycle_2 :- forward_rear_legs(forward_middle_legs).
; ditch_plan_cycle_2 :- forward_middle_legs(one_plan_cycle_done).


; ditch_plan_cycle_3 :- plan_state(start), retract(plan_state(start)), asserta(plan_stat
move_forward_front_legs)),
;                       move_forward_front_legs(move_forward_middle_legs).
; ditch_plan_cycle_3 :- move_forward_front_legs(move_back_middle_legs).
; ditch_plan_cycle_3 :- move_back_middle_legs(move_forward_rear_legs).
; ditch_plan_cycle_3 :- move_forward_rear_legs(one_plan_cycle_done).


; ditch_plan_cycle_4 :- plan_state(start), retract(plan_state(start)), asserta(plan_stat
move_forward_middle_legs)),
;                       move_forward_middle_legs(one_plan_cycle_done).
; ditch_plan_cycle_4 :- move_forward_middle_legs(one_plan_cycle_done).


; ditch_plan_cycle_5 :- plan_state(start), retract(plan_state(start)), asserta(plan_stat
move_forward_front_legs)),
```

```
;                        move_forward_front_legs(move_forward_middle_legs).
; ditch_plan_cycle_5 :- move_forward_front_legs(move_back_middle_legs).
; ditch_plan_cycle_5 :- move_back_middle_legs(move_forward_rear_legs).
; ditch_plan_cycle_5 :- move_forward_rear_legs(one_plan_cycle_done).


;
;***************  Cycles ****************
;


(defun initialize_plan_state ()
  (cond ((asserta plan_state 'start)
         t)
        (t nil)))


(defun ditch_plan_cycle_1 ()
  (cond ((and (match plan_state 'start)
              (retract plan_state 'start)
              (asserta plan_state 'place_legs_in_the_air)
              (place_legs_in_the_air 'back_middle_legs))
         t)
        ((place_legs_in_the_air 'back_middle_legs)
         t)
        ((back_middle_legs 'forward_rear_legs)
         t)
        ((forward_rear_legs 'forward_middle_legs)
         t)
        ((forward_middle_legs 'forward_front_legs)
         t)
        ((forward_front_legs 'lift_middle_legs_and_move)
         t)
        ((lift_middle_legs_and_move 'one_plan_cycle_done)
         t)
        (t nil)))


(defun ditch_plan_cycle_2()
  (cond ((and (match plan_state 'start)
              (retract plan_state 'start)
              (asserta plan_state 'back_middle_legs)
              (back_middle_legs 'forward_rear_legs))
         t)
        ((back_middle_legs 'forward_rear_legs)
         t)
        ((forward_rear_legs 'forward_middle_legs)
         t)
        ((forward_middle_legs 'one_plan_cycle_done)
         t)
        (t nil)))


(defun ditch_plan_cycle_3 ()
  (cond ((and (match plan_state 'start)
              (retract plan_state 'start)
              (asserta plan_state 'move_forward_front_legs)
              (move_forward_front_legs 'move_back_middle_legs))
         t)
        ((move_forward_front_legs 'move_back_middle_legs)
         t)
        ((move_back_middle_legs 'move_forward_rear_legs)
         t)
        ((move_forward_rear_legs 'one_plan_cycle_done)
```

```
        t)
       (t nil)))


(defun ditch_plan_cycle_4 ()
  (cond ((and (match plan_state 'start)
              (retract plan_state 'start)
              (asserta plan_state 'move_forward_middle_legs)
              (move_forward_middle_legs 'one_plan_cycle_done))
        t)
       ((move_forward_middle_legs 'one_plan_cycle_done)
        t)
       (t nil)))


(defun ditch_plan_cycle_5 ()
  (cond ((and (match plan_state 'start)
              (retract plan_state 'start)
              (asserta plan_state 'move_forward_front_legs)
              (move_forward_front_legs 'move_back_middle_legs))
        t)
       ((move_forward_front_legs 'move_back_middle_legs)
        t)
       ((move_back_middle_legs 'move_forward_rear_legs)
        t)
       ((move_forward_rear_legs 'one_plan_cycle_done)
        t)
       (t nil)))





;
;************** States ***************
;



; back_middle_legs(Next_State) :- plan_state(back_middle_legs), back_middle_legs_done,
;                                 retract(plan_state(back_middle_legs)), asserta(plan_st
e(Next_State)),
;                                 stop.
; back_middle_legs(Next_State) :- plan_state(back_middle_legs), do_back_middle_legs,
;                                 stop.


; forward_front_legs(Next_State) :- plan_state(forward_front_legs)), forward_front_legs_
ne,
;                                   retract(plan_state(forward_front_legs), asserta(plan
tate(Next_State)),
;                                   stop.
; forward_front_legs(Next_State) :- plan_state(forward_front_legs)), do_forward_front_le
,
;                                   stop.
;


; forward_middle_legs(Next_State) :- plan_state(forward_middle_legs), forward_middle_leg
```

```
done,
;                                   retract(plan_state(forward_middle_legs), asserta(p
_state(Next_State)),
;                                   stop.
; forward_middle_legs(Next_State) :- plan_state(forward_middle_legs), do_forward_middle_
gs,
;                                   stop.


; forward_rear_legs(Next_State) :- plan_state(forward_rear_legs), forward_rear_legs_done
;                                  retract(plan_state(forward_rear_legs), asserta(plan_s
te(Next_State)),
;                                  stop.
; forward_rear_legs(Next_State) :- plan_state(forward_rear_legs), do_forward_rear_legs,
;                                  stop.


; lift_middle_legs_and_move(Next_State) :- plan_state(lift_middle_legs_and_move), move_d
e, stop,
;                                          retract(plan_state(lift_middle_legs_and_move)
 asserta(plan_state(Next_State)).
; lift_middle_legs_and_move(Next_State) :- plan_state(lift_middle_legs_and_move), do_lif
middle_legs, move.
;


; move_back_middle_legs(Next_State) :- plan_state(move_back_middle_legs), move_back_midd
_legs_done,
;                                     retract(plan_state(move_back_middle_legs)), asser
(plan_state(Next_State)).
; move_back_middle_legs(Next_State) :- plan_state(move_back_middle_legs), do_move_back_m
dle_legs.


; move_forward_front_legs(Next_State) :- plan_state(move_forward_front_legs), move_forwa
_front_legs_done,
;                                       retract(plan_state(move_forward_front_legs)), a
erta(plan_state(Next_State)).
; move_forward_front_legs(Next_State) :- plan_state(move_forward_front_legs), do_move_fo
ard_front_legs.


; move_forward_middle_legs(Next_State) :- plan_state(move_forward_middle_legs), move_for
rd_middle_legs_done,
;                                        retract(plan_state(move_forward_front_legs)),
serta(plan_state(Next_State)).
; move_forward_middle_legs(Next_State) :- plan_state(move_forward_middle_legs), do_move_
rward_middle_legs.


; move_forward_rear_legs(Next_State) :- plan_state(move_forward_rear_legs), move_forwar
iddle_legs_done,
;                                       retract(plan_state(move_forward_rear_legs)), as
ta(plan_state(Next_State)).
; move_forward_rear_legs(Next_State) :- plan_state(move_forward_rear_legs), do_move_forw
d_rear_legs.
```

```
; place_legs_in_the_air(Next_State) :- plan_state(place_legs_in_the_air), place_legs_in_
e_air_done,
;                                     retract(plan_state(place_legs_in_the_air)), asser
(plan_state(Next_state)),
;                                     stop.
; place_legs_in_the_air(Next_State) :- plan_state(place_legs_in_the_air), do_place_legs_
_the_air, stop.



;
;*************** States ***************
;


(defun back_middle_legs (next_state)
  (cond ((and (match plan_state 'back_middle_legs)
              (back_middle_legs_done)
              (retract plan_state 'back_middle_legs)
              (asserta plan_state next_state)
              (stop))
         t)
        ((and (match plan_state 'back_middle_legs)
              (do_back_middle_legs)
              (stop))
         t)
        (t nil)))



(defun forward_front_legs (next_state)
  (cond ((and (match plan_state 'forward_front_legs)
              (forward_front_legs_done)
              (retract plan_state 'forward_front_legs)
              (asserta plan_state next_state)
              (stop))
         t)
        ((and (match plan_state 'forward_front_legs)
              (do_forward_front_legs)
              (stop))
         t)
        (t nil)))



(defun forward_middle_legs (next_state)
  (cond ((and (match plan_state 'forward_middle_legs)
              (forward_middle_legs_done)
              (retract plan_state 'forward_middle_legs)
              (asserta plan_state next_state)
              (stop))
         t)
        ((and (match plan_state 'forward_middle_legs)
              (do_forward_middle_legs)
              (stop))
         t)
        (t nil)))
```

```
(defun forward_rear_legs (next_state)
  (cond ((and (match plan_state 'forward_rear_legs)
              (forward_rear_legs_done)
              (retract plan_state 'forward_rear_legs)
              (asserta plan_state next_state)
              (stop))
         t)
        ((and (match plan_state 'forward_rear_legs)
              (do_forward_rear_legs)
              (stop))
         t)
        (t nil)))


(defun lift_middle_legs_and_move (next_state)
  (cond ((and (match plan_state 'lift_middle_legs_and_move)
              (move_done)
              (stop)
              (retract plan_state 'lift_middle_legs_and_move)
              (asserta plan_state next_state))
         t)
        ((and (match plan_state 'lift_middle_legs_and_move)
              (lift_middle_legs)
              (move))
         t)
        (t nil)))


(defun move_back_middle_legs (next_state)
  (cond ((and (match plan_state 'move_back_middle_legs)
              (move_back_middle_legs_done)
              (retract plan_state 'move_back_middle_legs)
              (asserta plan_state next_state))
         t)
        ((and (match plan_state 'move_back_middle_legs)
              (do_move_back_middle_legs))
         t)
        (t nil)))


(defun move_forward_front_legs (next_state)
  (cond ((and (match plan_state 'move_forward_front_legs)
              (move_forward_front_legs_done)
              (retract plan_state 'move_forward_front_legs)
              (asserta plan_state next_state))
         t)
        ((and (match plan_state 'move_forward_front_legs)
              (do_move_forward_front_legs))
         t)
        (t nil)))
```

```
(defun move_forward_middle_legs (next_state)
  (cond ((and (match plan_state 'move_forward_middle_legs)
              (move_forward_middle_legs_done)
              (retract plan_state 'move_forward_middle_legs)
              (asserta plan_state next_state))
         t)
        ((and (match plan_state 'move_forward_middle_legs)
              (do_move_forward_middle_legs))
         t)
        (t nil)))



(defun move_forward_rear_legs (next_state)
  (cond ((and (match plan_state 'move_forward_rear_legs)
              (move_forward_rear_legs_done)
              (retract plan_state 'move_forward_rear_legs)
              (asserta plan_state next_state))
         t)
        ((and (match plan_state 'move_forward_rear_legs)
              (do_move_forward_rear_legs))
         t)
        (t nil)))



(defun place_legs_in_the_air (next_state)
  (cond ((and (match plan_state 'place_legs_in_the_air)
              (place_legs_in_the_air_done)
              (retract plan_state 'place_legs_in_the_air)
              (asserta plan_state next_state)
              (stop))
         t)
        ((and (match plan_state 'place_legs_in_the_air)
              (do_place_legs_in_the_air)
              (stop))
         t)
        (t nil)))





;
;****************  State Executors ********************
;


; move_back_middle_legs_done :- all_middle_legs_lifted, all_middle_legs_placed,
;                               clear_middle_lifted_memory, clear_move_memory, stop.

; do_move_back_middle_legs :- all_middle_legs_lifted, move_done, stop, place_middle_leg
ack.
; do_move_back_middle_legs :- all_middle_legs_lifted, move.
; do_move_back_middle_legs :- lift_middle_legs, stop.
```

```
; move_forward_front_legs_done :- all_front_legs_lifted, all_front_legs_placed,
;                                 clear_front_lifted_memory, clear_move_memory, stop.

; do_move_forward_front_legs :- all_front_legs_lifted, move_done, stop, place_front_legs
; do_move_forward_front_legs :- all_front_legs_lifted, move.
; do_move_forward_front_legs :- lift_front_legs, stop.


; move_forward_middle_legs_done :- all_middle_legs_lifted, all_middle_legs_placed,
;                                  clear_middle_lifted_memory, clear_move_memory, stop.

; do_move_forward_middle_legs :- all_middle_legs_lifted, move_done, stop, place_middle_l
s.
; do_move_forward_middle_legs :- all_middle_legs_lifted, move.
; do_move_forward_middle_legs :- lift_middle_legs, stop.


; move_forward_rear_legs_done :- all_rear_legs_lifted, all_rear_legs_placed,
;                                clear_rear_lifted_memory, clear_move_memory, stop.

; do_move_forward_rear_legs :- all_rear_legs_lifted, move_done, stop, place_rear_legs.
; do_move_forward_rear_legs :- all_rear_legs_lifted, move.
; do_move_forward_rear_legs :- lift_rear_legs, stop.


; move :- asserta(resume_movement).
; stop :- asserta(stop_movement).


; clear_move_memory :- retract(move(done)).
; clear_move_memory.


; move_done :- move(done).
; move_done :- X is at_tkm_limit, X \== nil, asserta(move(done)).
; move_done :- X is at_stability_limit, X \== nil, asserta(move(done)).


; back_middle_legs_done :- all_middle_legs_lifted, all_middle_legs_placed, clear_middle_
fted_memory, clear_move_memory.
;
; do_back_middle_legs :- all_middle_legs_lifted, place_middle_legs_back.
; do_back_middle_legs :- lift_middle_legs.
;
; all_middle_legs_lifted :- middle_legs(lifted).
; all_middle_legs_lifted :- X is both_middle_legs_lifted, X == t, asserta(middle_legs(li
ed)).
;
; all_middle_legs_placed :- X is both_middle_legs_placed, X == t.
;
; clear_middle_lifted_memory :- retract(middle_legs(lifted)).
;
; place_middle_legs_back :- A_leg is placable_middle_leg, A_leg \== nil, asserta(decisio
A_leg,_,place_back)).
; place_middle_legs_back.
```

```
;
; lift_middle_legs :- A_leg is liftable_middle_leg, A_leg \== nil, asserta(decision(A_le
_,lift)).
; lift_middle_legs.
;



; forward_front_legs_done :- all_front_legs_lifted, all_front_legs_placed, clear_front_l
ted_memory, clear_move_memory.
;
; do_forward_front_legs :- all_front_legs_lifted, place_front_legs.
; do_forward_front_legs :- lift_front_legs.
;
; all_front_legs_lifted :- front_legs(lifted).
; all_front_legs_lifted :- X is both_front_legs_lifted, X == t, asserta(front_legs(lifte
).
;
; all_front_legs_placed :- X is both_front_legs_placed, X == t.
;
; clear_front_lifted_memory :- retract(front_legs(lifted)).
;
; place_front_legs :- A_leg is placable_front_leg, A_leg \== nil, asserta(decision(A_leg
,place)).
; place_front_legs.
;
; lift_front_legs :- A_leg is liftable_front_leg, A_leg \== nil, asserta(decision(A_leg,
lift)).
; lift_front_legs.



; forward_middle_legs_done :- all_middle_legs_lifted, all_middle_legs_placed, clear_midd
_lifted_memory, clear_move_memory.
;
; do_forward_middle_legs :- all_middle_legs_lifted, place_middle_legs.
; do_forward_middle_legs :- lift_middle_legs.
;
; place_middle_legs :- A_leg is placable_middle_leg, A_leg \== nil, asserta(decision(A_l
,_,place)).
; place_middle_legs.
;



; forward_rear_legs_done :- all_rear_legs_lifted, all_rear_legs_placed, clear_rear_lifte
memory, clear_move_memory.
;
; do_forward_rear_legs :- all_front_legs_lifted, place_rear_legs.
; do_forward_rear_legs :- lift_rear_legs.
;
; all_rear_legs_lifted :- rear_legs(lifted).
; all_rear_legs_lifted :- X is both_rear_legs_lifted, X == t, asserta(rear_legs(lifted))
;
; all_rear_legs_placed :- X is both_rear_legs_placed, X == t.
;
; clear_rear_lifted_memory :- retract(rear_legs(lifted)).
;
; place_rear_legs :- A_leg is placable_rear_leg, A_leg \== nil, asserta(decision(A_leg,_
lace)).
; place_rear_legs.
;
; lift_rear_legs :- A_leg is liftable_rear_leg, A_leg \== nil, asserta(decision(A_leg,_,
```

```
ft)).
; lift_rear_legs.
;



; do_lift_middle_legs :- lift_middle_legs.
;
;



; place_legs_in_the_air_done :- X is all_legs_placed, X == t.

; place_legs :- A_leg is placable_leg, A_leg \== nil,
;               asserta(decision(A_leg,_,place)).
; place_legs.



;
;****************  State Executors ********************
;


(defun move_back_middle_legs_done ()
  (cond ((and (all_middle_legs_lifted)
              (all_middle_legs_placed)
              (clear_middle_lifted_memory)
              (clear_move_memory)
              (stop))
         t)
        (t nil)))


(defun do_move_back_middle_legs ()
  (cond ((and (all_middle_legs_lifted)
              (move_done)
              (place_middle_legs_back)
              (stop))
         t)
        ((and (all_middle_legs_lifted)
              (move))
         t)
        ((and (lift_middle_legs)
              (stop))
         t)
        (t nil)))


(defun move_forward_front_legs_done ()
  (cond ((and (all_front_legs_lifted)
              (all_front_legs_placed)
              (clear_front_lifted_memory)
              (clear_move_memory)
```

```
                (stop))
         t)
        (t nil)))


(defun do_move_forward_front_legs ()
  (cond ((and (all_front_legs_lifted)
              (move_done)
              (place_front_legs)
              (stop))
         t)
        ((and (all_front_legs_lifted)
              (move))
         t)
        ((and (lift_front_legs)
              (stop))
         t)
        (t nil)))



(defun move_forward_middle_legs_done ()
  (cond ((and (all_middle_legs_lifted)
              (all_middle_legs_placed)
              (clear_middle_lifted_memory)
              (clear_move_memory)
              (stop))
         t)
        (t nil)))


(defun do_move_forward_middle_legs ()
  (cond ((and (all_middle_legs_lifted)
              (move_done)
              (place_middle_legs)
              (stop))
         t)
        ((and (all_middle_legs_lifted)
              (move))
         t)
        ((and (lift_middle_legs)
              (stop))
         t)
        (t nil)))



(defun move_forward_rear_legs_done ()
  (cond ((and (all_rear_legs_lifted)
              (all_rear_legs_placed)
              (clear_rear_lifted_memory)
              (clear_move_memory)
              (stop))
         t)
        (t nil)))
```

```
(defun do_move_forward_rear_legs ()
  (cond ((and (all_rear_legs_lifted)
              (move_done)
              (place_rear_legs)
              (stop))
         t)
        ((and (all_rear_legs_lifted)
              (move))
         t)
        ((and (lift_rear_legs)
              (stop))
         t)
        (t nil)))


(defun move ()
  (cond ((asserta resume_movement)
         t)
        (t nil)))


(defun stop ()
  (cond ((asserta stop_movement)
         t)
        (t nil)))


(defun clear_move_memory ()
  (cond ((retract move 'done)
         t)
        (t t)
        (t nil)))


(defun move_done ()
  (cond ((match move 'done)
         t)
        ((and (at_tkm_limit)
              (asserta move 'done))
         t)
        ((and (at_stability_limit)
              (asserta move 'done))
         t)
        (t nil)))


(defun back_middle_legs_done ()
  (cond ((and (all_middle_legs_lifted)
              (all_middle_legs_placed)
              (clear_middle_lifted_memory)
              (clear_move_memory))
```

```
        t)
       (t nil)))


(defun do_back_middle_legs ()
  (cond ((and (all_middle_legs_lifted)
              (place_middle_legs_back))
        t)
       ((lift_middle_legs)
        t)
       (t nil)))


(defun all_middle_legs_lifted ()
  (cond ((match middle_legs 'lifted)
        t)
       ((and (both_middle_legs_lifted)
             (asserta middle_legs 'lifted))
        t)
       (t nil)))


(defun all_middle_legs_placed ()
  (cond ((both_middle_legs_placed)
        t)
       (t nil)))


(defun clear_middle_lifted_memory ()
  (cond ((retract middle_legs 'lifted)
        t)
       (t nil)))


(defun place_middle_legs_back ()
  (let (leg)
    (cond ((and (setf leg (placable_middle_leg))
                (asserta decision (list 'place_back leg)))
           t)
          (t t)
          (t nil))))


(defun lift_middle_legs ()
  (let (leg)
    (cond ((and (setf leg (liftable_middle_leg))
                (asserta decision (list 'lift leg)))
           t)
          (t t)
          (t nil))))




(defun forward_front_legs_done ()
  (cond ((and (all_front_legs_lifted)
              (all_front_legs_placed)
              (clear_front_lifted_memory)
              (clear_move_memory))
        t)
       (t nil)))
```

```
(defun do_forward_front_legs ()
  (cond ((and (all_front_legs_lifted)
              (place_front_legs))
         t)
        ((lift_front_legs)
         t)
        (t nil)))


(defun all_front_legs_lifted ()
  (cond ((match front_legs 'lifted)
         t)
        ((and (both_front_legs_lifted)
              (asserta front_legs 'lifted))
         t)
        (t nil)))


(defun all_front_legs_placed ()
  (cond ((both_front_legs_placed)
         t)
        (t nil)))


(defun clear_front_lifted_memory ()
  (cond ((retract front_legs 'lifted)
         t)
        (t nil)))


(defun place_front_legs ()
  (let (leg)
    (cond ((and (setf leg (placable_front_leg))
                (asserta decision (list 'place leg)))
           t)
          (t t)
          (t nil))))


(defun lift_front_legs ()
  (let (leg)
    (cond ((and (setf leg (liftable_front_leg))
                (asserta decision (list 'lift leg)))
           t)
          (t t)
          (t nil))))





(defun forward_middle_legs_done ()
  (cond ((and (all_middle_legs_lifted)
              (all_middle_legs_placed)
              (clear_middle_lifted_memory)
              (clear_move_memory))
         t)
        (t nil)))
```

```
(defun do_forward_middle_legs ()
  (cond ((and (all_middle_legs_lifted)
              (place_middle_legs))
         t)
        ((lift_middle_legs)
         t)
        (t nil)))


(defun place_middle_legs ()
  (let (leg)
    (cond ((and (setf leg (placable_middle_leg))
                (asserta decision (list 'place leg)))
           t)
          (t t)
          (t nil))))









(defun forward_rear_legs_done ()
  (cond ((and (all_rear_legs_lifted)
              (all_rear_legs_placed)
              (clear_rear_lifted_memory)
              (clear_move_memory))
         t)
        (t nil)))


(defun do_forward_rear_legs ()
  (cond ((and (all_rear_legs_lifted)
              (place_rear_legs))
         t)
        ((lift_rear_legs)
         t)
        (t nil)))


(defun all_rear_legs_lifted ()
  (cond ((match rear_legs 'lifted)
         t)
        ((and (both_rear_legs_lifted)
              (asserta rear_legs 'lifted))
         t)
        (t nil)))


(defun all_rear_legs_placed ()
  (cond ((both_rear_legs_placed)
         t)
        (t nil)))


(defun clear_rear_lifted_memory ()
  (cond ((retract rear_legs 'lifted)
         t)
        (t nil)))
```

```
(defun place_rear_legs ()
  (let (leg)
    (cond ((and (setf leg (placable_rear_leg))
                (asserta decision (list 'place leg)))
           t)
          (t t)
          (t nil))))


(defun lift_rear_legs ()
  (let (leg)
    (cond ((and (setf leg (liftable_rear_leg))
                (asserta decision (list 'lift leg)))
           t)
          (t t)
          (t nil))))









(defun do_lift_middle_legs ()
  (cond ((lift_middle_legs)
         t)
        (t nil)))









(defun place_legs_in_the_air_done ()
  (cond ((all_legs_placed)
         t)
        (t nil)))


(defun do_place_legs_in_the_air()
  (let (leg)
    (cond ((and (setf leg (placable_leg))
                (asserta decision (list 'place leg)))
           t)
          (t t)
          (t nil))))
```

```
;*****************************************************************
;
;  Plan Libraries
;
;*****************************************************************


; body_plan :- speed_plan, trajectory_plan.

; speed_plan :- retract(reduce_speed), slow_down.
; speed_plan :- speed_up.

; speed_up :- X is speed_up_robot.

; slow_down :- X is slow_down_robot.

; trajectory_plan :- stable_m, restore_trajectory.
; trajectory_plan :- modify_trajectory.

; stable_m :- Condition is stable_p_m, Condition == t.

; restore_trajectory :- X is restore_command.

; modify_trajectory :- X is modify_command.

; generate_decision :- retract(decision(A_leg,B_leg,A_decision)),
;                       X is send_decision(A_leg,B_leg,A_decision), fail.
; generate_decision :- retract(limit_leg(A_leg,A_decision)),
;                       X is send_decision(A_leg,_,A_decision), fail.
; generate_decision.



(defun body_plan()
  (cond ((and (speed_plan)
              (trajectory_plan))
         t)
        (t nil)))


(defun speed_plan()
  (cond ((and (retract reduce_speed)
              (slow_down))
         t)
        ((and (retract stop_movement)
              (stop_motion))
         t)
        ((and (retract resume_movement)
              (resume_motion)
              (speed_up))
         t)
        ((speed_up) t)
        (t nil)))


(defun speed_up()
  (cond (t (speed_up_robot) t)
        (t nil)))

(defun slow_down()
```

```
  (cond (t (slow_down_robot) t)
        (t nil)))


(defun trajectory_plan()
; OR
  (cond ((and (stable_m)
              (restore_trajectory))
         t)
        ((modify_trajectory) t)
        (t nil)))

(defun stable_m ()
 .(cond ((stable_p_m) t)
        (t nil)))


(defun restore_trajectory()
  (cond (t (restore_command) t)
        (t nil)))

(defun modify_trajectory()
  (cond (t (modify_command) t)
        (t nil)))


;(defun generate_decision()
;  (cond ((not (unify decision a-decision))
;         nil)
;        ((and (unify decision a-decision)
;              (retract decision a-decision)
;              (print (list (second a-decision) (third decision) (first decision)))
;              (send_decision (second a-decision) (thirdd decision) (first decision))
;              (generate_decision))
;         t)))


(defun generate_decision()
  (cond ((and decision
              (not
                (dolist (a-decision decision) (send-one-decision a-decision)))
                   ; dolist returns nil
              (not (setf decision nil))
              nil)  ; this simulates fail
         t)
        ((and limit_leg
              (unify limit_leg decision1)
              (send-one-decision decision1)
              (retract limit_leg '?)
              nil)
         t)
        (t t)
        (t nil)))



(defun send-one-decision (decision)
; format (decision leg1 leg2)
; lisp function
  (cond ((equal (first decision) 'exchange)
         (send_decision (second decision) (third decision) (first decision)))
        (t
         (send_decision (second decision) nil (first decision))))
  t)
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*******************************************************
;
;  sensor flavor definition
;
;*******************************************************

(defflavor sensor(state owner)
           ()
  :initable-instance-variables)



;*******************************************************
;
;  contact-sensor flavor definition
;
;*******************************************************

(defflavor contact-sensor()
           (sensor)
  :initable-instance-variables)


(defmethod (contact-sensor :initti)
           (leg-name)
  (setf state (send self :sensing)))


(defmethod (sensor :contact-p)
           ()
  state)


(defmethod (sensor :sensing)
           ()
; simulation purpose
; graph-terrain is object.
  (setf state
        (let* ((leg-pos-wrt-body (send (send owner :executor)
                                       :leg-pos-wrt-body))
               (leg-pos-wrt-earth
                 (to-earth-transform (send owner :get-H1) leg-pos-wrt-body))
               (x-y-pos (list (first leg-pos-wrt-earth)
                              (second leg-pos-wrt-earth)))
               (leg-height (third leg-pos-wrt-earth)))
          (if (< leg-height (+ 0.07 (send graph-terrain :get-height x-y-pos)))
              t
              nil))))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

;*****************************************************
;
;  stability-calculator flavor definition
;
;*****************************************************

(defflavor stability-calculator(safety-margin
                                safety-margin-p
                                large-safety-margin
                                large-safety-margin-p
                                recovery-vector
                                recovery-vector-p
                                owner)
          ()
  :initable-instance-variables)


(defmethod (stability-calculator :initti)
          ()
  (setf safety-margin 0.4)
  (setf safety-margin-p 0.2)
  (setf large-safety-margin 0.5)
  (setf large-safety-margin-p 0.4)
  (setf recovery-vector '(0 0 0))
  (setf recovery-vector-p '(0 0 0)))


(defmethod (stability-calculator :get-recovery-vector)
          ()
  recovery-vector)


(defmethod (stability-calculator :get-recovery-vector-p)
          ()
  recovery-vector-p)


(defmethod (stability-calculator :convert-to-recovery-vector)
          (stability-vector)
  (let ((sm (first stability-vector))
        (vect (second stability-vector)))
    (cond ((< sm 0)
           nil)
          ((< sm 0.1)
           (magvect (/ 1 sm) vect))
          (t
           (magvect (/ 0.1 (* sm sm)) vect)))))


(defmethod (stability-calculator :more-stable)
          (supporting-legs H leg1 leg2)
  (let ((stability1 (send self :calculate-stability
                                    (cons leg1 supporting-legs) H))
        (stability2 (send self :calculate-stability
                                    (cons leg2 supporting-legs) H)))
    (if (> stability1 stability2)
        t
        nil)))
```

```
(defmethod (stability-calculator :stable-m)
; predict H <= H10
          (supporting-legs H)
  (let ((stability-vector
          (send self :get-stability supporting-legs H)))
    (cond ((>= (first stability-vector)
               large-safety-margin)
           t)
          (t
           (if (>= (first stability-vector) safety-margin)
               (setf recovery-vector
                     (send self :convert-to-recovery-vector stability-vector))
               (setf recovery-vector '(0 0 0)))
           nil))))


(defmethod (stability-calculator :stable-p-m)
; present H <= H1
          (supporting-p-legs H)
  (let* ((stability-vector
          (send self :get-stability supporting-p-legs H))
         (st-margin (first stability-vector)))
    (cond ((>= st-margin
               large-safety-margin-p)
           t)
          (t
           (setf recovery-vector-p
                 (send self :convert-to-recovery-vector stability-vector))
           (if (< st-margin safety-margin-p)
               (my-print (list 'st-p st-margin)))
           nil))))


(defmethod (stability-calculator :stable)
          (supporting-legs H10)
  (if (>= (send self :calculate-stability
                                 supporting-legs H10)
          safety-margin)
      t
      nil))


(defmethod (stability-calculator :stable-p)
          (supporting-p-legs H1)
  (if (>= (send self :calculate-stability
                                 supporting-p-legs H1)
          safety-margin-p)
      t
      nil))


(defmethod (stability-calculator :calculate-stability)
          (supporting-legs H)
  (first (send self :get-stability supporting-legs H)))


(defmethod (stability-calculator :get-stability)
```

```
            (supporting-legs H)
  (if (>= (counting supporting-legs) 3)
      (measure-distance (center-of-gravity H)
                        (convex-hull
                          (supporting-points
                            supporting-legs)))
      '(-100.0 (0 0 0))))


(defun convex-hull (points)
; returns clockwise-ordered
; point list of convex hull
   (reverse
     (convex1 (car points) points
            '(0 0 0) nil)))


(defun convex1 (current-point points previous-pt visited-pts)
  (let* ((min-out-pt (min-out current-point previous-pt points))
         (pos (position min-out-pt visited-pts :test #'equal)))
    (cond (pos
           (subseq visited-pts 0 (+ pos 1)))
          (t (convex1 min-out-pt
                      points
                      current-point
                      (cons min-out-pt visited-pts))))))


(defun min-out (current-pt pv-pt pts)
  (let* ((min-pt nil)
         (min-angle 100)
         (angle 0))
    (dolist (a-pt pts)
      (cond ((not (equal a-pt current-pt))
             (setf angle (turning-angle (vectsub current-pt pv-pt)
                                        (vectsub a-pt current-pt)))
             (cond ((< angle min-angle)
                    (setf min-pt a-pt)
                    (setf min-angle angle))))))
    min-pt))


(defun turning-angle(vect new-vect)
; 2 D space clock-wise turning angle
; Neither vect should not be zero vector.
  (let* ((vect1-0 (list (first vect) (second vect) 0))
         (vect2-0 (list (first new-vect) (second new-vect) 0))
         (normal-vect (crossprod vect1-0 vect2-0))
         (polarity (> (third normal-vect) 0))
         (value (/ (dotprod vect1-0 vect2-0)
                   (* (magnitude vect1-0)
                      (magnitude vect2-0))))
         (angle 0))
    (if (>= value 1)
        (setf value 1))
    (if (<= value -1)
        (setf value -1))
```

```
    (setf angle (acos value))
    (if polarity
        (- (* 2 pi) angle)
        angle)))



(defun center-of-gravity (H)
; center-of-body is represented wrt earth coordinate.
  (let ((x (aref H 0 3))
        (y (aref H 1 3)))
    (list x y)))
; center-of-body can be changed in future.



(defun find-slope (first-point second-point)
  (let ((del-x (- (car  second-point) (car  first-point)))
        (del-y (- (cadr second-point) (cadr first-point))))
    (if (> (abs del-x) 0.0000001)
        (/ del-y del-x)
        nil)))



(defun infinite-case (x a-line)
  (list x
        (+ (* (car a-line) x) (cadr a-line))))


(defun intersection-point (a-line b-line)
; Returns list (x y).      Line is list (slope  crossing-point-of-axis).
  (cond ((null (car a-line)) (infinite-case (cadr a-line) b-line))
        ((null (car b-line)) (infinite-case (cadr b-line) a-line))
        (t  (normal-case a-line b-line))))


(defun in-side-of-convex-hull (center-point first-points second-points)
  (do* ((first-points  first-points  (cdr first-points))
        (second-points second-points (cdr second-points))
        (in-side-flag T))
       ((null first-points) in-side-flag)
    (if (test-out-side (car first-points) center-point (car second-points))
        (setf in-side-flag nil))))


(defun line (slope point)
  (if slope
      (list slope (- (second point) (* slope (first point))))
      (list slope (first point))))
; When slope is infinitive, return with x-axis crossing point instead of
; y-axis crossing point.


(defun measure-distance (center-point  convex-points)
```

```
; convex-points is a list of points
; point is a list (x y z).
  (let* ((first-points convex-points)
         (second-points (append (cdr convex-points)
                 (list (car first-points)))))
    (if (in-side-of-convex-hull center-point first-points second-points)
        (start-measure center-point first-points second-points)
        '(-10.0 (0 0 0)))))
                 ; center-of-gravity is out-side of support pattern


(defun normal-case (a-line b-line)
  (let* ((a1 (car a-line))
         (b1 (cadr a-line))
         (a2 (car b-line))
         (b2 (cadr b-line))
         (x  (/ (- b1 b2) (- a2 a1)))
         (y  (+ (* a1 x) b1)))
    (list x y)))


(defun point-distance (center-point first-point second-point)
; returns distance and vector between cross-pt and center-pt
  (let* ((slope1 (find-slope first-point second point))
         (slope2 (right-angle slope1))
         (cross-pt (intersection-point (line slope1 first-point)
                                       (line slope2 center-point)))
         (del-vect (vectsub center-point cross-pt ))
         (distance (magnitude del-vect)))
    (list distance (list (first del-vect) (second del-vect) 0.0))))


(defun right-angle (slope)
  (cond ((null slope) 0.0)              ; infinitive input slope
        ((< (abs slope) 0.0000001) nil) ; zerop slope
        (t (/ (- 1) slope))))


(defun start-measure (center-point first-points second-points)
  (do* ((first-points first-points (cdr first-points))
        (second-points second-points (cdr second-points))
        (min-distance 10000.0 min-distance)   ; infinte dummy number 10000.0
        (min-direction nil) (dis-dir nil))
       ((null first-points)  (list min-distance min-direction))
    (setf dis-dir (point-distance center-point
                                  (car first-points) (car second-points)))
    (cond ((< (first dis-dir) min-distance)
           (setf min-distance (first dis-dir))
           (setf min-direction (second dis-dir))))))


(defun supporting-points (legs)
  (mapcar #'(lambda (leg)
              (send leg :foothold))
          legs))
```

```
(defun test-out-side (first-point second-point third-point)
  (let* ((a (- (cadr first-point) (cadr third-point)))
         (b (- (car  third-point) (car  first-point)))
         (c (- (+ (* a (car third-point)) (* b (cadr third-point)))))
         (decision (+ (* a (car  second-point))
                      (* b (cadr second-point))
                      c)))
    (if (>= decision 0.0)
        T
        nil)))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
; stop-body definition
;
;*****************************************************************


(defflavor stop-body (stop-body-motion-flag)
           (body)
  :initable-instance-variables)


(defmethod (stop-body :after :initti)
           ()
  (setf stop-body-motion-flag nil))


(defmethod (stop-body :stop-body-motion)
           ()
  (setf stop-body-motion-flag t))


(defmethod (stop-body :restore-body-motion)
           ()
  (setf stop-body-motion-flag nil))



(defmethod (stop-body :calculate-motion)
           (joystick-command legs)
  (setf joy-command joystick-command)
  (cond ((equal support-plane-clock 10)
         (setf estimated-support-plane
               (send support-plane-estimator :get-plane legs))
         (setf support-plane-clock 0)))
  (setf support-plane-clock (+ support-plane-clock 1))
  (cond
    ((or stop-motion-flag stop-body-motion-flag (null modify-vector-p))
     (send body-controller :control
              '(0 0 0)
              0 estimated-support-plane))
    (modify-vector-p
     (send body-controller :control
              (vectadd joy-command (send self :get-modify-vector-p))
              deceleration-factor estimated-support-plane))
    (t
     (control body-controller
              (vectadd joy-command (send self :get-modify-vector))
              deceleration-factor estimated-support-plane))))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*******************************************************
;
;  support-plane-estimator flavor definition
;
;*******************************************************


(defflavor support-plane-estimator(owner)
           ()
  )



(defmethod (support-plane-estimator :initti)
           ()
  )



(defmethod (support-plane-estimator :get-plane)
           (legs)
  (let* ((footholds-for-estimation (get-footholds legs))
         (constants (get-constants footholds-for-estimation)))
      (make-plane-from-coefficient constants)))




;*******************************************************
;
;  support-plane-estimator.get-plane
;
;*******************************************************



(defun add-points (points)
; returns a list (number-of-points  sum-of-points).
  (do ((points points (cdr points))
       (i 0 (+ i 1))
       (sum-vect '(0 0 0)))
      ((null points) (list i sum-vect))
    (setf sum-vect (vectadd (car points) sum-vect))))



(defun average-point (points)
  (let* ((num-&-sum-vect (add-points points))
         (number-of-points (first num-&-sum-vect))
         (sum-vect (second num-&-sum-vect)))
    (if (> number-of-points 0)
        (magvect (/ 1 number-of-points) sum-vect)
        (print "Error in finding average-point of estimate plane"))))


(defun get-a0 (bar-point a1)
  (let* ((x-bar (first bar-point))
         (z-bar (third bar-point)))
    (- z-bar (* a1 x-bar))))
```

```
(defun get-a1 (points bar-point common-denominator)
; returns a1 which is sum in this function.
  (do* ((points points (cdr points))
        (sum 0)
        (x nil)  (x-bar (first bar-point))
        (z nil)  (z-bar (third bar-point)))
       ((null points) (/ sum common-denominator))
    (setf x (first (car points)))
    (setf z (third (car points)))
    (setf sum (+ sum (* (- x x-bar) (- z z-bar))))))


(defun get-a2 (points bar-point common-denominator)
; returns a2 which is sum in this function.
  (do* ((points points (cdr points))
        (sum 0)
        (x nil)  (x-bar (first  bar-point))
        (y nil)  (y-bar (second bar-point)))
       ((null points) (/ sum common-denominator))
    (setf x (first  (car points)))
    (setf y (second (car points)))
    (setf sum (+ sum (* (- x x-bar) (- y y-bar))))))


(defun get-a3 (bar-point a2)
  (let* ((x-bar (first  bar-point))
         (y-bar (second bar-point)))
    (- y-bar (* a2 x-bar))))


(defun get-a4 (points a0 a1 a2 a3)
  (let* ((number-of-points (counting points))
         (yr (get-yr points a2 a3))
         (zr (get-zr points a0 a1))
         (yr-bar (get-yr-bar yr number-of-points))
         (zr-bar (get-zr-bar zr number-of-points)))
    (do ((yr     yr     (cdr yr))
         (zr     zr     (cdr zr))
         (numerator 0) (a-yr 0) (a-zr 0)
         (denominator 0))
        ((null yr) (/ numerator denominator))
      (setf a-yr (first yr))
      (setf a-zr (first zr))
      (setf numerator   (+ numerator   (* (- a-yr yr-bar) (- a-zr zr-bar))))
      (setf denominator (+ denominator (* (- a-yr yr-bar) (- a-yr yr-bar)))))))


(defun get-common-denominator (points bar-point)
  (do* ((points points (cdr points))
        (sum 0)
        (x nil)
        (x-bar (first bar-point)))
       ((null points) sum)
    (setf x (first (car points)))
    (setf sum (+ sum (* (- x x-bar) (- x x-bar))))))
```

```
(defun get-constants (points)
  (let* ((bar-point (average-point points))
         (common-denominator (get-common-denominator points bar-point))
         (a1 (get-a1 points bar-point common-denominator))
         (a2 (get-a2 points bar-point common-denominator))
         (a0 (get-a0 bar-point a1))
         (a3 (get-a3 bar-point a2))
         (a4 (get-a4 points a0 a1 a2 a3)))
    (list a0 a1 a2 a3 a4)))


(defun get-footholds (legs)
  (do* ((legs legs (cdr legs))
        (footholds nil)
        (a-leg nil))
       ((null legs) footholds)
    (setf a-leg (car legs))
    (if (send a-leg :foothold)
        (setf footholds (cons (send a-leg :foothold) footholds)))))


(defun get-yr (points a2 a3)
  (do* ((points points (cdr points))
        (yr nil)
        (x  nil)
        (y  nil))
       ((null points) (reverse yr))
    (setf x (first  (car points)))
    (setf y (second (car points)))
    (setf yr (cons (- y a2 (* a3 x)) yr))))


(defun get-yr-bar (yr number-of-points)
  (do ((yr yr (cdr yr))
       (yr-bar 0))
      ((null yr) (/ yr-bar number-of-points))
    (setf yr-bar (+ yr-bar (first yr)))))


(defun get-zr (points a0 a1)
  (do* ((points points (cdr points))
        (zr nil)
        (x  nil)
        (z  nil))
       ((null points) (reverse zr))
    (setf x (first  (car points)))
    (setf z (third  (car points)))
    (setf zr (cons (- z a0 (* a1 x)) zr))))


(defun get-zr-bar (zr number-of-points)
  (do ((zr zr (cdr zr))
       (zr-bar 0))
      ((null zr) (/ zr-bar number-of-points))
    (setf zr-bar (+ zr-bar (first zr)))))
```

```
(defun make-plane-from-coefficient (constants)
  (let* ((a0 (first  constants))
         (a1 (second constants))
         (a2 (third  constants))
         (a3 (fourth  constants))
         (a4 (fifth  constants))
         (a  (- (* a4 a3) a1))
         (b  (- a4))
         (c  1)
         (d  (- (* a2 a4) a0))
         (unit-normal (normalize-vector (list a b c)))
         (dis (/ d (magnitude (list a b c)))))
    (list unit-normal dis)))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************
;
;  terrain-regulator flavor definition
;
;*****************************************************


(defflavor terrain-regulator(body-rotate-rate-x body-rotate-rate-y
                             body-trans-rate-z  old-body-rotate-rate-x
                             old-body-rotate-rate-y old-body-trans-rate-z
                             gain min-height max-height
                             eta1 eta2 min-eta max-eta)
           (regulator)
  :initable-instance-variables)



(defmethod (terrain-regulator :initti)
           ()
  (setf gain 5)

  (setf min-eta 0.0000001)              ; 0   degree
  (setf max-eta 0.4363)                 ; 25  degrees
  (setf min-height 4.4)                 ; 4.4 feet
  (setf max-height 5.4)                 ; 5.4 feet
  (setf eta1 min-eta)                   ; 0   degree
  (setf eta2 0.5236)                    ; 30  degree

  (setf body-rotate-rate-x 0.0)
  (setf body-rotate-rate-y 0.0)
  (setf body-trans-rate-z 0.0)
  (list body-rotate-rate-x body-rotate-rate-y body-trans-rate-z))


(defmethod (terrain-regulator :do-terrain-regulation)
           (k-gamma-delta-height)
; k-gamma-delta-height is ((k.x k.y k.z) gamma delta-height).
  (let* ((k     (first  k-gamma-delta-height))
         (gamma (second k-gamma-delta-height))
         (delta-height (third k-gamma-delta-height))
         (body-rotate-rate-x-n (* gain (first  k) gamma))
         (body-rotate-rate-y-n (* gain (second k) gamma))
         (body-trans-rate-z-n  (* gain delta-height)))
    (setf body-rotate-rate-x
          (send self :limitor
            (send self :filter body-rotate-rate-x-n body-rotate-rate-x)
            0.1))
    (setf body-rotate-rate-y
          (send self :limitor
            (send self :filter body-rotate-rate-y-n  body-rotate-rate-y)
            0.1))
    (setf body-trans-rate-z
          (send self :limitor
            (send self :filter body-trans-rate-z-n body-trans-rate-z)
            1)))
  (list body-rotate-rate-x body-rotate-rate-y body-trans-rate-z))



(defmethod (terrain-regulator :eta-function)
```

```
            (eta)
  (let ((slope (/ (- max-eta min-eta) (- eta2 eta1))))
    (+ min-eta (* slope (- eta eta1)))))



(defmethod (terrain-regulator :get-k-gamma-by-slope)
            (plane H)
  (let* ((plane-rpt-body (plane-transform plane H))
         (height (cadr plane-rpt-body))
         (eta (arc-cos (third (car plane))))
         (k-gamma-desired-height nil))
    (setf k-gamma-desired-height
          (cond ((< eta eta1) (send self :low-slope plane))
                ((< eta eta2) (send self :mid-slope eta plane H))
                (T            (send self :high-slope plane H))))
    (list (first  k-gamma-desired-height)
          (second k-gamma-desired-height)
          (- (third k-gamma-desired-height) height))))



(defmethod (terrain-regulator :height-function)
            (eta)
  (let ((slope (/ (- max-height min-height) (- eta2 eta1))))
    (- max-height (* slope (- eta eta1)))))



(defmethod (terrain-regulator :high-slope)
            (plane H)
  (let* ((plane-unit-normal (first plane))
         (a (first  plane-unit-normal))
         (b (second plane-unit-normal))
         (m (sqrt (+ (* a a) (* b b))))
         (desired-eta max-eta)
         (desired-height min-height)
         (desired-body-plane (list (list (* (/ a m) (sin desired-eta))
                                         (* (/ b m) (sin desired-eta))
                                         (cos desired-eta)) 0.0))
         (desired-body-plane-in-body (plane-transform desired-body-plane H))
         (unit-normal-body-plane (first desired-body-plane-in-body))
         (a1 (first  unit-normal-body-plane))
         (b1 (second unit-normal-body-plane))
         (c1 (third  unit-normal-body-plane))
         (m1 (sqrt (+ (* a1 a1) (* b1 b1))))
         (k  (if (= m1 0)
                 (list 0 0 0)
                 (list (/ (- b1) m1) (/ a1 m1) 0)))
         (gamma (arc-cos c1)))
    (list k gamma desired-height)))



(defmethod (terrain-regulator :limitor)
            (vel max-vel)
  (if (>= (abs vel) max-vel)
      (if (> vel 0)
          max-vel
          (- max-vel))
      vel))
```

```
(defmethod (terrain-regulator :low-slope)
           (plane)
  (let* ((unit-normal (first plane))
         (a (first  unit-normal))
         (b (second unit-normal))
         (c (third  unit-normal))
         (m (sqrt (+ (* a a) (* b b))))
         (k.a nil)
         (k.b nil)
         (gamma (arc-cos c))
         (desired-height max-height))
    (if (= m 0.0)
        (setf k.a 0.0          k.b 0.0)
        (setf k.a (/ (- b) m)  k.b (/ a m)))
    (list (list k.a k.b 0.0) gamma desired-height)))


(defmethod (terrain-regulator :mid-slope)
           (eta plane H)
  (let* ((plane-unit-normal (first plane))
         (a (first  plane-unit-normal))
         (b (second plane-unit-normal))
         (m (sqrt (+ (* a a) (* b b))))
         (desired-eta (send self :eta-function eta))
         (desired-height (send self :height-function eta))
         (desired-body-plane (list (list (* (/ a m) (sin desired-eta))
                                         (* (/ b m) (sin desired-eta))
                                         (cos desired-eta)) 0.0))
         (desired-body-plane-in-body (plane-transform desired-body-plane H))
         (unit-normal-body-plane (first desired-body-plane-in-body))
         (a1 (first  unit-normal-body-plane))
         (b1 (second unit-normal-body-plane))
         (c1 (third  unit-normal-body-plane))
         (m1 (sqrt (+ (* a1 a1) (* b1 b1))))
         (k  (if (= m1 0)
                 (list 0 0 0)
                 (list (/ (- b1) m1) (/ a1 m1) 0)))
         (gamma (arc-cos c1)))
    (list k gamma desired-height)))


(defmethod (terrain-regulator :regulate)
           (estimated-support-plane H)
  (let ((k-gamma (send self :get-k-gamma-by-slope estimated-support-plane H)))
    (send self :do-terrain-regulation k-gamma)))


(defmethod (terrain-regulator :restore)
           ()
  (setf body-rotate-rate-x old-body-rotate-rate-x)
  (setf body-rotate-rate-y old-body-rotate-rate-y)
  (setf body-trans-rate-z  old-body-trans-rate-z)
  (list body-rotate-rate-x body-rotate-rate-y body-trans-rate-z))


(defmethod (terrain-regulator :save)
           ()
  (setf old-body-rotate-rate-x body-rotate-rate-x)
```

```
  (setf old-body-rotate-rate-y body-rotate-rate-y)
  (setf old-body-trans-rate-z  body-trans-rate-z)
  )
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
; test-overlap-leg definition
;
;*****************************************************************


(defflavor test-overlap-leg ()
           (overlap-leg)
  )



(defmethod (test-overlap-leg :change-to-back-foothold)
           ()
  (setf foothold (first foothold-list))
  (setf tkm (first tkm-list)))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  test-overlap-robot definition
;
;*****************************************************************


(defflavor test-overlap-robot ()
           (overlap-robot)
  )


(defmethod (test-overlap-robot :initti)
           ()
  (send graph-asv :init-data)
  (setf vision-system (make-instance 'ditch-vision-system :owner self))
  (send vision-system :initti)
  (setf joystick (make-instance 'joystick))
  (send joystick :reset)
  (empty-queue lift-queue)
  (setf lift-flag t)
  (let ((H))
    (setf body (make-instance 'body :owner self))
    (setf H (send body :initti))
    (setf legs (list
                 (make-instance 'test-overlap-leg :name 'leg1  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg2  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg3  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg4  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg5  :owner self)
                 (make-instance 'test-overlap-leg :name 'leg6  :owner self)
                 ))
    (mapcar #'(lambda (a-leg) (send a-leg :initti H)) legs))
  )



(defmethod (test-overlap-robot :send-decision)
           (leg1 leg2 a-decision)
  (cond ((equal a-decision 'exchange)
         (send leg1 :send-decision a-decision)
         (send leg2 :send-decision 'place)
         (send leg1 :send-exchange leg2))
        ((equal a-decision 'place_back)
         (send leg1 :change-to-back-foothold)
         (send leg1 :send-decision 'place))
        (t
         (send leg1 :send-decision a-decision))))



(defmethod (test-overlap-robot :has-more-tkm)
           (leg1 leg2)
  (let ((tkm1 (send leg1 :tkm))
        (tkm2 (send leg2 :tkm)))
    (cond ((null tkm2)
           t)
          ((null tkm1)
           nil)
          ((> tkm1
              tkm2)))))
```

```
(defmethod (test-overlap-robot :liftable-leg)
           (leg)
  (cond ((member leg lift-able-legs :test #'equal)
         leg)
        (t nil)))

(defmethod (test-overlap-robot :placable-leg)
           (leg)
  (cond ((and (member leg place-able-legs :test #'equal)
              (send leg :has-foothold-p))
         leg)
        (t nil)))



(defmethod (test-overlap-robot :liftable-front-leg)
           ()
  (cond ((send self :liftable-leg (first legs)))
        ((send self :liftable-leg (second legs)))
        (t nil)))

(defmethod (test-overlap-robot :placable-front-leg)
           ()
  (cond ((send self :placable-leg (first legs)))
        ((send self :placable-leg (second legs)))
        (t nil)))

(defmethod (test-overlap-robot :both-front-legs-placed)
           ()
; If one of front legs has not foothold, then this will fail!
  (cond ((and (member (first legs) supporting-p-legs :test #'equal)
              (member (second legs) supporting-p-legs :test #'equal))
         t)
        (t nil)))

(defmethod (test-overlap-robot :both-front-legs-lifted)
           ()
  (cond ((and (not (member (first legs) supporting-p-legs :test #'equal))
              (not (member (second legs) supporting-p-legs :test #'equal)))
         t)
        (t nil)))



(defmethod (test-overlap-robot :liftable-rear-leg)
           ()
  (cond ((send self :liftable-leg (fifth legs)))
        ((send self :liftable-leg (sixth legs)))
        (t nil)))
```

```
(defmethod (test-overlap-robot :placable-rear-leg)
           ()
  (cond ((send self :placable-leg (fifth legs)))
        ((send self :placable-leg (sixth legs)))
        (t nil)))


(defmethod (test-overlap-robot :both-rear-legs-placed)
           ()
; If one of rear legs has not foothold, then this will fail!
  (cond ((and (member (fifth legs) supporting-p-legs :test #'equal)
              (member (sixth legs) supporting-p-legs :test #'equal))
         t)
        (t nil)))


(defmethod (test-overlap-robot :both-rear-legs-lifted)
           ()
  (cond ((and (not (member (fifth legs) supporting-p-legs :test #'equal))
              (not (member (sixth legs) supporting-p-legs :test #'equal)))
         t)
        (t nil)))





(defmethod (test-overlap-robot :liftable-middle-leg)
           ()
  (cond ((send self :liftable-leg (third legs)))
        ((send self :liftable-leg (fourth legs)))
        (t nil)))


(defmethod (test-overlap-robot :placable-middle-leg)
           ()
  (cond ((send self :placable-leg (third legs)))
        ((send self :placable-leg (fourth legs)))
        (t nil)))


(defmethod (test-overlap-robot :both-middle-legs-placed)
           ()
; If one of middle legs has not foothold, then this will fail!
  (cond ((and (member (third legs) supporting-p-legs :test #'equal)
              (member (fourth legs) supporting-p-legs :test #'equal))
         t)
        (t nil)))


(defmethod (test-overlap-robot :both-middle-legs-lifted)
           ()
  (cond ((and (not (member (third legs) supporting-p-legs :test #'equal))
              (not (member (fourth legs) supporting-p-legs :test #'equal)))
         t)
        (t nil)))



(defmethod (test-overlap-robot :one-placable-leg)
```

```
            ()
  (cond ((send self :placable-front-leg))
        ((send self :placable-middle-leg))
        ((send self :placable-rear-leg))
        (t nil)))


(defmethod (test-overlap-robot :all-legs-placed)
            ()
  (cond ((and (send self :both-front-legs-placed)
              (send self :both-middle-legs-placed)
              (send self :both-rear-legs-placed))
         t)
        (t nil)))



(defmethod (test-overlap-robot :at-ditch-area)
            ()
  (send vision-system :on-ditch-area
        (send self :get-H10)))





;*************************************************************
;
;
;   Prolog Interface Functions
;
;
;*************************************************************

(defun liftable_front_leg ()
  (send asv :liftable-front-leg))


(defun placable_front_leg ()
  (send asv :placable-front-leg))


(defun both_front_legs_placed ()
  (send asv :both-front-legs-placed))


(defun both_front_legs_lifted ()
  (send asv :both-front-legs-lifted))


(defun liftable_rear_leg ()
  (send asv :liftable-rear-leg))


(defun placable_rear_leg ()
  (send asv :placable-rear-leg))


(defun both_rear_legs_placed ()
  (send asv :both-rear-legs-placed))
```

```
(defun both_rear_legs_lifted ()
  (send asv :both-rear-legs-lifted))

(defun liftable_middle_leg ()
  (send asv :liftable-middle-leg))

(defun placable_middle_leg ()
  (send asv :placable-middle-leg))

(defun both_middle_legs_placed ()
  (send asv :both-middle-legs-placed))

(defun both_middle_legs_lifted ()
  (send asv :both-middle-legs-lifted))


(defun placable_leg ()
  (send asv :one-placable-leg))

(defun all_legs_placed ()
  (send asv :all-legs-placed))

(defun at_ditch_area ()
  (send asv :at-ditch-area))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************
;
;  tkm-calculator flavor definition
;
;*****************************************************


(defflavor tkm-calculator(working-volume owner)
           ()
  :initable-instance-variables)


(defmethod (tkm-calculator :initti)
           (leg-name)
  (cond ((equal leg-name 'leg1)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -8.0832) ((0 0.9397  0.3420) -2.569))
                 (((0 0 1) 5.7313) ((1 0 0) -3.4167) ((0 0.9397 -0.3420) -2.569)))))
        ((equal leg-name 'leg2)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -8.0832) ((0 0.9397  0.3420)  2.569))
                 (((0 0 1) 5.7313) ((1 0 0) -3.4167) ((0 0.9397 -0.3420)  2.569)))))
        ((equal leg-name 'leg3)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -2.2915) ((0 0.9397  0.3420) -2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  2.2915) ((0 0.9397 -0.3420) -2.569)))))
        ((equal leg-name 'leg4)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0) -2.2915) ((0 0.9397  0.3420)  2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  2.2915) ((0 0.9397 -0.3420)  2.569)))))
        ((equal leg-name 'leg5)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0)  3.3332) ((0 0.9397  0.3420) -2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  7.8332) ((0 0.9397 -0.3420) -2.569)))))
        ((equal leg-name 'leg6)
         (setf working-volume
               '((((0 0 1) 3.316)  ((1 0 0)  3.3332) ((0 0.9397  0.3420)  2.569))
                 (((0 0 1) 5.7313) ((1 0 0)  7.8332) ((0 0.9397 -0.3420)  2.569)))))
        )
  )


(defmethod (tkm-calculator :find-tkm)
           (a-foothold body-trans-rate body-rotate-rate)
; a-foothold is based on body coordinate
; returns tkm
  (let* ((leg-vel-rpt-body
           (get-leg-velocity
             a-foothold body-trans-rate body-rotate-rate)))
    (get-tkm a-foothold leg-vel-rpt-body working-volume)))


(defun get-distance (planes velocity leg-position)
; global function : plane-distance
; before start, make one plane list
  (do ((planes (append (first planes) (second planes)) (cdr planes))
       (a-tkm nil)
       (min-tkm 10000))
      ((null planes) min-tkm)
    (setf a-tkm (plane-distance (car planes) velocity leg-position))
```

```
    (if a-tkm
        (if (and (> a-tkm 0) (> min-tkm a-tkm))
            (setf min-tkm a-tkm)))))


(defun get-leg-velocity (pos-rpt-body body-trans-rate body-rotate-rate)
; returns leg-velocity-wrt-body
;         = - ( body-trans-rate + body-rotate-rate X pos-rpt-body )
  (vectsub '(0 0 0)
           (vectadd body-trans-rate
                    (crossprod body-rotate-rate pos-rpt-body))))


(defun get-tkm (leg-pos-rpt-body velocity working-volume)
; global function : magnitude
; outside w.v returns nil.  If speed is near 0, then returns 1000.0.
  (if (in-side-volume leg-pos-rpt-body working-volume)
      (let ((speed (magnitude velocity)))
        (if (< speed 1/1000)
            1000.0
            (/ (get-distance working-volume velocity leg-pos-rpt-body) speed)))
      nil))


(defun in-side-volume (position planes)
; planes ((up front left) (back right bottom))
  (let* ((positive-planes (first planes))
         (negative-planes (second planes))
         (inside-flag T))
    (dolist (a-plane positive-planes)
      (if (>= (plane-normal-distance a-plane position) 0)
          (setf inside-flag nil)))
    (dolist (a-plane negative-planes)
      (if (<= (plane-normal-distance a-plane position) 0)
          (setf inside-flag nil)))
    inside-flag))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  user interface routines
;
;*****************************************************************


(defvar *old-terrain-file-name*)
(defvar *new-terrain-file-name*)
(defvar *terrain-slope-type*)
(defvar *terrain-slope-data*)
(defvar *terrain-slope-angle*)
(defvar *terrain-type*)
(defvar *obstacle-ratio*)
(defvar *random-seed*)
(defvar *ok-flag*)
(defvar *screen*)
(defvar *screen-width*)
(defvar *screen-height*)
(defvar *new-lisp-listener*)
(defvar *ditch-width*)
(defvar *ditch-location*)
(defvar *ditch-type*)


(defun initialize-menu-variables ()
  (setf *old-terrain-file-name* nil)
  (setf *new-terrain-file-name* nil)
  (setf *terrain-slope-type* 'default)
  (setf *terrain-slope-data* nil)
  (setf *terrain-slope-angle* 0)
  (setf *terrain-type* 'random)
  (setf *obstacle-ratio* '25)
  (setf *random-seed* '125)
  (setf *ok-flag* t)
  (setf *ditch-width* 6)
  (setf *ditch-location* 21)
  (setf *ditch-type* 'no-ditch))


(defun get-old-terrain-file-name()
  (let ((file-names
          (mapcar #'(lambda (file)
                      (list (file-namestring file) ':documentation "Use an old terrain")
                    (directory "robot:kwak.robot.terrain-data;*.*"))))
    (if file-names
        (w:menu-choose (cons '("new-terrain" :value nil :documentation "Create a new ter
in")
                             file-names)
                       :label "Select terrain"
                       :superior *new-lisp-listener*)
        (w:menu-choose '(("new-terrain" :value nil :documentation "Create a new terrain"
                       :label "Select terrain"
                       :superior *new-lisp-listener*))))


(defun get-terrain-slope-type()
  (w:choose-variable-values
    '((*terrain-slope-type* :menu-alist (("Default" :value default)
                                         ("Single Angle" :value single-angle)
                                         ("Manual" :value manual))))
```

```
    :label "Choose terrain slope profile"
    :superior *new-lisp-listener*)
  *terrain-slope-type*)


(defun get-terrain-slope-angle()
  (w:choose-variable-values
    '((*terrain-slope-angle* :documentation "Input terrain slope angle"
                             :constraint (lambda (d1 d2 d3 value)
                                           (cond ((> (abs value) 30) "Too steep angle")
                                                 (t nil)))))
    :label "Input terrain slope angle"
    :superior *new-lisp-listener*)
  *terrain-slope-angle*)


(defun get-terrain-slope-data()
  (setf *terrain-slope-data* '((15 0) (30 2)))
  (w:choose-variable-values
    '((*terrain-slope-data* :documentation "Input format ((x1 h1) (x2 h2) ... )"
                            :constraint (lambda (d1 d2 d3 value)
                                          (cond ((null value) "Please input slope")
                                                (t nil)))))
    :label "Input slope data"
    :superior *new-lisp-listener*)
  *terrain-slope-data*)


(defun get-terrain-obstacle-type ()
  (w:choose-variable-values
    '((*terrain-type* :menu-alist (("Random" :value random)
                                   ("Manual" :value manual))))
    :label "Choose type of terrain"
    :superior *new-lisp-listener*)
  *terrain-type*)


(defun get-terrain-random-data()
  (w:choose-variable-values
    '((*obstacle-ratio* :constraint (lambda (d1 d2 d3 value)
                                      (cond ((> value 90) "Too Big")
                                            ((< value 0) "Error")
                                            (t nil))))
              (*random-seed* :fixnum))
    :superior *new-lisp-listener*)
  (list *obstacle-ratio* *random-seed*))


(defun get-ditch-type ()
  (w:choose-variable-values
    '((*ditch-type* :menu-alist (("Add Ditch" :value add-ditch)
                                 ("No Ditch" :value no-ditch))))
    :label "Choose ditch option"
    :superior *new-lisp-listener*)
  *ditch-type*)
```

```
(defun get-ditch-width-location()
  (w:choose-variable-values
    '((*ditch-width* :constraint (lambda (d1 d2 d3 value)
                                   (cond ((> value 7) "Too Big")
                                         ((< value 3) "Too Small")
                                         (t nil))))
      (*ditch-location* :constraint (lambda (d1 d2 d3 value)
                                   (cond ((> value 32) "Too Big")
                                         ((< value 15) "Too Small")
                                         (t nil)))))
    :superior *new-lisp-listener*)
  (list *ditch-width* *ditch-location*))



(defun user-ok()
  (setf *ok-flag* t)
    (w:choose-variable-values
      '((*ok-flag* :boolean))
        :label "Do you like this terrain?"
        :superior *new-lisp-listener*)
    *ok-flag*)



(defun user-file-name()
  (w:choose-variable-values
    '((*new-terrain-file-name* :string))
    :label "Please provide the output file name."
    :superior *new-lisp-listener*)
  *new-terrain-file-name*)



(defun user-save()
  (setf *ok-flag* nil)
  (w:choose-variable-values
    '((*ok-flag* "Save-p" :boolean))
    :label "Do you want to save this terrain?"
    :superior *new-lisp-listener*)
  *ok-flag*)



(defun move-and-shape-lisp-listener()
  (setf *screen* (send *terminal-io* :superior))
  (setf *screen-width* (send *screen* :width))
  (setf *screen-height* (send *screen* :height))
  (setf *new-lisp-listener* (make-instance 'w:lisp-listener))
  (send *new-lisp-listener* :refresh)
  (send *new-lisp-listener* :set-size
        (truncate (* 1.0 *screen-width*)) (truncate (* 0.2 *screen-height*)))
  (send *new-lisp-listener* :set-position
        0 (truncate (* 0.8 *screen-height*)))
  (send *new-lisp-listener* :set-more-p nil)
  (send *new-lisp-listener* :select))
```

```
(defun restore-lisp-listener()
  (send *new-lisp-listener* :kill))


(defun my-print (x)
  (print x *new-lisp-listener*))


(defun my-read-char-no-hang ()
  (read-char-no-hang *new-lisp-listener*))
```

```
28  13
(1        6.625   0.0      3.0)
(2        6.625   0.0      1.08)
(3        6.625  -2.0      1.08)
(4       -6.625  -2.0      1.08)
(5       -6.625   2.0      1.08)
(6        6.625   2.0      1.08)
(7        6.625   0.9     -3.1)
(8        6.625  -0.9     -3.1)
(9       -6.625  -0.9     -3.1)
(10      -6.625   0.9     -3.1)
(11       0.0     0.0      0.0)
(12       0.0     0.0      0.0)
(13       0.0     0.0      0.0)
(14       0.0     0.0      0.0)
(15       0.0     0.0      0.0)
(16       0.0     0.0      0.0)
(17       0.0     0.0      0.0)
(18       0.0     0.0      0.0)
(19       0.0     0.0      0.0)
(20       0.0     0.0      0.0)
(21       0.0     0.0      0.0)
(22       0.0     0.0      0.0)
(23       0.0     0.0      0.0)
(24       0.0     0.0      0.0)
(25       0.0     0.0      0.0)
(26       0.0     0.0      0.0)
(27       0.0     0.0      0.0)
(28       0.0     0.0      0.0)
(2 1 2)
(5 3 4 5 6 3)
(2 6 7)
(2 5 10)
(2 4 9)
(2 3 8)
(5 8 7 10 9 8)
(3 11 12 13)
(3 14 15 16)
(3 17 18 19)
(3 20 21 22)
(3 23 24 25)
(3 26 27 28)
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;
;  vision-system definition

;
;*****************************************************************


(defflavor vision-system (owner)
           ()
  :initable-instance-variables)


(defmethod (vision-system :initti)
           ()
  )


(defmethod (vision-system :scanning)
           ()
  )


(defmethod (vision-system :permitted-cell)
           (t-cell)
  (send graph-terrain :permitted-cell t-cell))


(defmethod (vision-system :terrain-point)
           (t-cell)
  (send graph-terrain :terrain-point t-cell))
```

Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Library, Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 4401 Ford Avenue Alexandria, Virginia 22302-0268	1
Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Department Chairman, Code CSMz Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
Professor Neil C. Rowe, Code CSRp Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
Professor Michael J. Zyda, Code CSZk Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
Professor Yuh-Jeng Lee, Code CSLe Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
Professor Se-Hung Kwak, Code CSKw Department of Computer Science Naval Postgraduate School Monterey, CA 93943	2

Professor Kenneth J. Waldron 1
Department of Mechanical Engineering
Ohio State University
206 West 18th Avenue
Columbus, Ohio 43210

Professor D.E. Orin 1
Department of Electrical Engineering
Ohio State University
2015 Neil Avenue
Columbus, Ohio 43210

Col. Eric Mettala 1
DARPA/ISTO
1400 Wilson Boulevard
Arlington, Virginia 22209

Doctor Robert Rosenfeld 1
DARPA/AUSTO
1555 Wilson Boulevard, Suite 600
Arlington, Virginia 22209