INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.

2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.

3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*

4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.

UMI Dissertation Information Service
University Microfilms International
A Bell & Howell Information Company
300 N. Zeib Road, Ann Arbor, Michigan 48106
Kwak, Se-Hung

A COMPUTER SIMULATION STUDY OF A FREE GAIT MOTION COORDINATION ALGORITHM FOR ROUGH-TERRAIN LOCOMOTION BY A HEXAPOD WALKING MACHINE

The Ohio State University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark \( \checkmark \).

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background \( \checkmark \)
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages \( \checkmark \)
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print _____
11. Page(s) ________ lacking when material received, and not available from school or author.
12. Page(s) ________ seem to be missing in numbering only as text follows.
13. Two pages numbered ______. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received \( \checkmark \)
16. Other

__________________________________________________________

__________________________________________________________

University
Microfilms
International
A Computer Simulation Study of a Free Gait Motion Coordination Algorithm for Rough-Terrain Locomotion by a Hexapod Walking Machine

A Dissertation
Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of the Ohio State University

by

Se-Hung Kwak, B.S.E.E., M.S.E.E.

* * * *

The Ohio State University
1986

Dissertation Committee:
Professor C. A. Klein
Professor R. B. McGhee
Professor D. E. Orin

Approved by:

Advisor
Department of Electrical Engineering
ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my advisor Professor Robert B. McGhee for his constant motivation, support, and encouragement throughout this work. I am also grateful to Professor Charles A. Klein, and Professor David E. Orin for their valuable contributions and for their review of this work.

I am indebted to others in the ASV project who have made valuable contributions to this work for careful discussion and valuable suggestions during the development of this free gait motion coordination algorithm. Also, I would like to thank Ms. Debra Britton and Ms. Jacqueline Buckner for their excellent work in preparation of this manuscript.

Finally, I thank my parents and each of my family members for their support and encouragement. Especially, I would like to thank my wife, Myungwon, for her assistance, patience, and understanding in spite of her busy daily work with our two sons.

This work was sponsored in part by the Defense Advanced Research Projects Agency under Contract DAAAEE 07-84-K-R001, and in part by Ohio State University through funds provided by the Taine G. McDougal Professorship in Engineering Endowment. The computer simulation results provided in Chapter 7 of this dissertation were obtained with a Symbolics 3640 LISP machine made available to the author through the courtesy of Mr. Russell Davis, Directorate for Infor-
mation Management, U.S. Army Combat Developments Experimentation Center, Fort Ord, California. These sources of support are gratefully acknowledged.
To my family
VITA

May 29, 1956 ...................... Born - Seoul, Korea

1979 .................................. B.S.E.E., Electronics Engineering,
Seoul National University,
Seoul, Korea

1984 .................................. M.S.E.E., Electrical Engineering,
The Ohio State University,
Columbus, Ohio

1982-1986  ......................... Graduate Research Associate,
Digital Systems Laboratory,
The Ohio State University,
Columbus, Ohio

FIELDS OF STUDY

Major Field: Electrical Engineering

Studies in Computer Engineering: Professors R.B. McGhee,
K.W. Olson, D.E. Orin,
F. Ozguner

Studies in Control Engineering: Professors H. Ilemami,
R.J. Mayhan, U. Ozguner

Studies in Computer and Information Science: Professor V. Ashok,
K.Y. Lee, W.F. Ogden

Studies in Communication Engineering: Professor R.T. Compton
D.T. Davis
# TABLE OF CONTENTS

## ACKNOWLEDGEMENTS

ii

## VITA

v

## LIST OF FIGURES

x

## LIST OF TABLES

xiii

1. Introduction 1

   1.1 Background ................................. 1

   1.2 Organization .............................. 3

2. Survey of Previous Work 5

   2.1 Introduction .............................. 5

   2.2 Walking Machines ......................... 5

   2.3 Stepping Algorithms ........................ 12

      2.3.1 Stability ............................. 13

      2.3.2 Periodic Gaits ....................... 15

      2.3.3 Free Gaits ............................ 17

   2.4 Programming Paradigms and Languages .............. 20

   2.5 Summary .................................... 23
### 3. Detailed Problem Statement

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Vehicle Model</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1 Body Model</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2 Leg Model</td>
<td>30</td>
</tr>
<tr>
<td>3.3 Terrain Model</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Sensor Model</td>
<td>41</td>
</tr>
<tr>
<td>3.5 Human Interface</td>
<td>42</td>
</tr>
<tr>
<td>3.6 Summary</td>
<td>44</td>
</tr>
</tbody>
</table>

### 4. Control of Individual Legs

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>46</td>
</tr>
<tr>
<td>4.2 Leg Modeling</td>
<td>47</td>
</tr>
<tr>
<td>4.2.1 Leg Trajectory</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2 Foot Velocity Control State Transitions</td>
<td>52</td>
</tr>
<tr>
<td>4.3 Pascal Implementation</td>
<td>54</td>
</tr>
<tr>
<td>4.4 LISP Implementation</td>
<td>56</td>
</tr>
<tr>
<td>4.5 Prolog Implementation</td>
<td>59</td>
</tr>
<tr>
<td>4.6 OPS5 Implementation</td>
<td>63</td>
</tr>
<tr>
<td>4.7 Smalltalk Implementation</td>
<td>68</td>
</tr>
<tr>
<td>4.8 Summary</td>
<td>74</td>
</tr>
</tbody>
</table>

### 5. Body Control

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>78</td>
</tr>
<tr>
<td>5.2 Command Velocity Regulation</td>
<td>79</td>
</tr>
<tr>
<td>5.3 Terrain Adaptation</td>
<td>82</td>
</tr>
<tr>
<td>5.3.1 Automatic Body Regulation</td>
<td>82</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

2.1 Supervisory control scheme used with OSU Hexapod vehicle [8] .. 8
2.2 Logical decomposition of computational tasks in the ASV [14] .. 11
3.1 Kinematic simulation model of the Adaptive Suspension Vehicle . 27
3.2 Body geometry of the model vehicle ................................. 28
3.3 Geometry of the base plate and its foot position ................. 31
3.4 Cross-section of working volume in $Y_F - Z_F$ plane .......... 35
3.5 Working volume of ASV leg ........................................... 36
3.6 Constrained working volume of ASV leg ......................... 37
3.7 Typical simulation terrain with the model vehicle .............. 39
4.1 Leg motion relative to body during forward body motion over level terrain .............................................................. 48
4.2 Foot velocity control state diagram for free gait algorithm. (All states command leg rates using actual leg position feedback.) .... 53
5.1 Command velocity regulation scheme ............................... 80
5.2 Two extremes of body control ....................................... 84
5.3 Body altitude and attitude control strategy .................... 85
6.1 Overall logical hierarchy of the free gait simulation program. (There exists a one second time difference between the planners and the execution blocks.) ................................. 92
6.2 Full size reachable areas of six legs and initial sixteen possible footholds ................................................................. 99

6.3 Leg motion planning state diagram for free gait algorithm. (Planner runs 1 sec. in future. Assumes current joystick input remains constant for 1 sec. in to future.) ............................................................. 106

6.4 Complete velocity control regulation scheme ................................................................. 114

6.5 Flowchart for free gait simulation program ................................................................. 116

6.6 Flowchart for plan-motion block ........................................................................ 118

6.7 Flowchart of lift-exchange block ........................................................................ 122

6.8 Flowchart of try-exchange block ........................................................................ 124

6.9 Flowchart of exchange block ........................................................................ 126

6.10 Flowchart of place block ........................................................................ 127

6.11 Flowchart of slow-down-vehicle-or-use-new-footholds block .................. 129

6.12 Flowchart of try-new-foothold block ........................................................................ 131

6.13 Flowchart of deadlock-recovery block ........................................................................ 132

7.1 Standard trajectory for the model vehicle for evaluation of performance in the presence of forbidden cells. (Velocity components of each segment of trajectory are shown at the end of the figure.) ................................................................. 149

7.2 Typical random terrain example ........................................................................ 151

7.3 Experimental results for the algorithm on randomly generated terrain with zero-angle slope along the standard path ........................................................................ 152

7.4 Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when the safety margin is 0.5 feet. ................................................................. 154
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5</td>
<td>Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when the safety margin is 0.9 feet.</td>
<td>155</td>
</tr>
<tr>
<td>7.6</td>
<td>Experimental results for the algorithm on randomly generated terrain with zero angle slope along a straight line when the safety margin is 0.9 feet.</td>
<td>156</td>
</tr>
<tr>
<td>7.7</td>
<td>Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when one of the front legs is inoperative.</td>
<td>160</td>
</tr>
<tr>
<td>7.8</td>
<td>Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when one of the middle legs is inoperative.</td>
<td>161</td>
</tr>
<tr>
<td>7.9</td>
<td>Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when one of the rear legs is inoperative.</td>
<td>162</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Maximal magnitudes of each velocity component commanded by an 
operator ................................................................. 43

7.1 Timing analysis of the simulation program ......................... 144
CHAPTER 1

Introduction

1.1 Background

Until recently, all land vehicles have been supported and propelled by wheels or tracks. These vehicles usually provide very efficient means of transportation on favorable surfaces, such as roads, with respect to speed, maneuverability, and fuel efficiency. However, for off-road applications, the capabilities of such vehicles are sharply reduced. In contrast, animals, and human beings, which utilize legs as the fundamental means to transport themselves, have no great difficulties in traveling over unprepared surfaces. In fact, on rough terrain, animals can move much faster while consuming significantly less energy than conventional wheeled or tracked vehicles [1]. Specifically, though a wheel or track sinks into soft soil and creates a depression which it is continuously trying to climb out of, legs use only discrete footprints in which any back slip pushes up soil material behind the foot which helps to increase traction [2]. Unfortunately, this type of terrain can be frequently found. One study reveals that roughly half of the land surface of the earth is not accessible to conventional wheeled or tracked vehicles [3].

The usage of legged vehicles is not limited to the above mentioned rough terrain. They are applicable to tasks which are dangerous for human beings, but which cannot be carried out by conventional wheeled or tracked vehicles, such as underwater exploration, fire fighting, harvesting trees, underground mining, and
nuclear power plant inspection and repair.

From the above discussion, it is clear that legged locomotion provides three advantages over wheels or tracks for off-road transportation: increased speed, improved fuel economy, and greater mobility [4]. However, despite the apparent advantages of legged locomotion, man-made legged vehicles have not yet been used to any great extent because a legged vehicle offers too many degrees-of-freedom which must be independently controlled. Experiments have shown that this coordination task is too difficult to be handled well by a human operator [4].

Because of advances in electronics, especially the development of powerful microcomputers, it is now feasible to consider the construction of computer-controlled walking vehicles, and several experimental machines of this type have been built [4-7]. A very basic problem in the control of walking vehicles is the selection of an appropriate gait which is the particular sequence used for limb cycling during locomotion. Until now, almost all such research has dealt with periodic gaits in which the order of foot placing and lifting events is fixed in advance. While such gaits are entirely satisfactory for straight-line locomotion at constant speed, they are less well adapted to complex maneuvering and for extreme terrain conditions[8-11]. This dissertation presents a simulation study of algorithms for on-line optimization of stepping patterns leading to non-periodic or free gaits [10,11]. The work presented allows for the possibility of a terrain microstructure in which some terrain cells are suitable for load bearing while others are not [10,11]. In addition, unlike most prior work, no restrictions are imposed on the direction of the body motion; i.e., body control is omni-directional. Also, the model terrain is not limited to flat terrain, but is generalized to three-dimensional terrain.

In order to conduct this study in a realistic context, one walking vehicle is considered as a model vehicle, the Adaptive Suspension Vehicle (ASV). This vehicle
has been built at Ohio State University [12], and is currently under test. It is the author's hope that the algorithm developed in this research can eventually be tested on this machine.

1.2 Organization

A brief summary of previous research work on legged vehicles relating to this study is given in Chapter 2. This includes a historical review of various walking machines developed during the last twenty years, stepping algorithms, stability problems, and gaits. Additionally, a survey of programming paradigms and programming languages is included.

In Chapter 3, a detailed problem statement is provided. This chapter presents the geometry of the walking vehicle model, the terrain model used in this simulation study, the sensor models utilized by the model vehicle, and the means of human interaction for simulation experiments.

Chapter 4 describes the leg modeling for the rest of the study. This modeling includes the trajectory of the foot, and the description of the foot velocity control finite-state machine. Following this discussion, a finite-state machine for leg velocity control is implemented using five different programming languages. Based on a comparison study of the five programs written in the five languages, one language (LISP) is suggested as the programming language to implement the algorithm developed in this study.

Chapter 5 presents the body motion control portion of the algorithm of this dissertation. Specifically, this chapter includes the command velocity regulation scheme, and the terrain adaptation method.

In Chapter 6, the specific free gait motion coordination algorithm of this dissertation is discussed. This chapter covers the foothold selection method, the
leg motion planning finite-state machine which directly controls the foot velocity control finite-state machine discussed in Chapter 4, and the body deceleration method. Finally, using all the concepts developed in this dissertation, the free gait algorithm is presented with the aid of hierarchically organized flowcharts.

In Chapter 7, first, the LISP implementation of the free gait algorithm is described, and then the algorithm is evaluated in various ways. A timing analysis, a statistical evaluation of the performance on terrain with forbidden cells, a steep slope performance evaluation, and an analysis of the behavior of the algorithm in the presence of one or two leg failures are performed.

Finally, Chapter 8 summarizes the algorithms developed in this research and suggests areas where further work could be done. The appendices at the end of this dissertation are divided into two sections. The first section includes the program listings discussed in Chapter 4, and the second section lists the free gait program which is written in LISP for a Symbolics 3640 LISP machine.
CHAPTER 2

Survey of Previous Work

2.1 Introduction

This chapter gives an overview of previous work performed in areas related to this dissertation work, and also presents some background information and definitions. In the following section, a brief history of major walking machines which are important to this study is presented. After this survey, earlier several stepping algorithms are described, and a number of terms relating to vehicle stability are defined. In Section 2.4, programming paradigms and languages are presented as background information for Chapter 4.

2.2 Walking Machines

During the past two decades, several experimental walking machines have been introduced. The first machine capable of active accommodation of terrain irregularities was the Quadruped Transporter built by the General Electric Company [4]. This machine walked successfully in 1968, and demonstrated some of the potential advantages of walking machines. Its physical size was approximately comparable to that of an elephant and its weight was about 3000 pounds. It could handle a payload of about 500 pounds and could walk up to 2 miles per hour. In order to solve the coordination problems of this vehicle, natural human locomotion abilities were used to maintain the balance of the vehicle and to manage the forces
applied to each of the legs. Specifically, a control lever was assigned to each of the
four legs of the vehicle. Then, when operating this machine, a human operator,
who rode in it, controlled the four levers using his two hands and two feet. Each
lever had three degrees of freedom corresponding to two degrees of freedom at the
hip of each vehicle leg and one degree of freedom at its knee. In order to provide
the human operator with a feeling of the interaction between the ground and sup­
porting feet, a force feedback servomechanism which reflected approximately one
percent of actual vehicle joint torques into the operator's control levers was imple­
mented. All of these functions were realized with hydraulic technology, without
the use of any electric or electronic components [4].

Even though the G.E. Quadruped Transporter was successful in walking on
soft soils, traversing rough terrain, and surmounting obstacles, it turned out that
several practical problems limited its usefulness. One of the most serious of these
problems was the inefficiency of the hydraulic system. Because of the hydraulic
configuration utilized one hydraulic power supply for all actuators, the system was
always operated at near its maximum hydraulic pressure. Therefore, the hydraulic
configuration of the vehicle was not adequate, in the sense of fuel economy, to be
utilized for a self-contained walking machine.

The other practical problem was the difficulty of limb motion coordination.
Actually, the machine was so demanding that it was practically impossible to
control it continuously for a long period of time [4]. Apparently, the 12 degrees
of freedom involved to control the machine substantially exceed the 6 degrees of
freedom which is ordinarily considered as the maximum number that an average
person can handle, as for example, in controlling a helicopter. To solve this prob­
lem, body control, in which only the movement of the center of a machine and
the orientation of its body are controlled by a human operator and coordination
of each leg motion is performed by a computer, might be introduced rather than limb level control.

The above concept to automate limb control was already started at the same period of time, and introduced the first computer-controlled walking machine in 1966 [5]. This machine was called the “Phoney Pony”. It had four legs, and could walk along only in a straight-line over level terrain. The limb motion coordination of this machine was accomplished by a specially designed hard-wired digital computer. Later, the concept of this machine led to much improved control mechanism in which body attitude is explicitly controlled, known as supervisory control [8,9].

The first successful walking machine utilizing supervisory control is the OSU Hexapod (Ohio State University Hexapod). The supervisory control scheme used with this system is illustrated in Fig. 2.1. As shown in Fig. 2.1, in this approach, the human operator receives desired tasks and then makes use of visual feedback information to determine the interaction between terrain and the vehicle. Using this information, he provides steering, speed, and mode commands to the vehicle. The motion planning software block, using these commands, determines the desired trajectory of the body and legs of the vehicle. Specifically, in order to achieve a desired trajectory, a gait pattern, estimated body altitude and attitude over the terrain, and desired foot velocities and forces are calculated in the motion planning block [8,9]. The motion execution block generates actuator commands for leg joints by comparing actual foot positions, rates, and forces to those desired. Thus, the OSU Hexapod is a fully computer controlled walking machine, but it is not a self-contained machine. Rather it communicates with a PDP-11/70 minicomputer through a trailing cable pulled by the Hexapod, and is also electrically powered through the cable in order to operate electric motors attached to leg joints and the machine electronics, such as sensors, A/D, and D/A convertors.
Figure 2.1: Supervisory control scheme used with OSU Hexapod vehicle [8]
The first self-contained walking machine was the Carnegie-Mellon University Six-Legged Walking Machine, which took its first steps in 1982 [13]. The vehicle was big enough to carry a human operator, and had a gasoline-powered hydraulic system to manipulate its six legs. An on-board computer controlled leg position, and leg sequencing by controlling a total of 36 hydraulic valves (corresponding to six valves for each of the six legs). In order to manipulate the valves of the six legs, the program inside the computer maintained six independently running finite-state machines. But most of the leg motions needed for walking were controlled by hydraulic circuits without either human or computer intervention. This machine was controlled in speed and direction by a human operator (for body level control), but used no sophisticated servos to provide proportional control of limb forces and movement.

In 1985, the Adaptive Suspension Vehicle (ASV) was completed at Ohio State University. This machine was under test during the writing of this dissertation. The physical dimensions of the ASV [12,14] are 17 feet in length, 7.5 feet in width, and 8 feet nominal height at the back of the vehicle body. Its total weight is about 7,000 lb., including driver, optical terrain scanner mounted on the top of the cab, fuel and hydraulic fluids, and a 500 lb. payload in its internal cargo bay. The detailed geometry of this vehicle will be described in Chapter 3 because this vehicle is used as the vehicle model for the research of this dissertation. Though this vehicle does not operate fully autonomously, it is controlled in a supervisory control mode by a human operator with aids from advanced sensing, computer processing, and coordination systems. Historically, the ASV is the first machine to utilize an optical terrain scanner, which is a three-dimensional optical sensor using the time-of-flight principle for range measurement. This optical sensor and other types of sensors employed in the vehicle will be described in more detail in
Chapter 3.

The ASV operational modes are composed of a total of six modes, which are: utility, precision footing, close maneuvering, terrain-following, cruise, and dash [14]. Among the six modes, the terrain-following and cruise modes utilize the optical scanner, but the cruise mode will be the most efficient mode in the sense of fuel economy. The dash mode is intended to achieve the designed top speed of 8 mph.

In order to effectively control the vehicle, the computer system of the ASV is designed with a network of microcomputers which has the on-board computer architecture as shown in Fig. 2.2 [14]. The partitioning in Fig. 2.2 shows the logical decomposition of the computational tasks, and physically the microcomputers are also grouped in the way as shown in Fig. 2.2, in order to minimize the data communication among the physical computers. For example, one leg control computer is realized by one Intel iSBC 86/30 single-board computer with the 8087 floating-point coprocessor as well as 16 channels of analog input and eight channels of analog output lines. On the other hand, the guidance computer is realized with four iSBC 86/30 computers, and the coordination computer is realized with two computers. The legs of the ASV are made of aluminum box beams, and actuated by hydraulic actuators in order to achieve adequate force and speed capabilities. Overall, the ASV uses nineteen pumps, eighteen servo-valves, eighteen rotary actuators, and eighteen linear actuators.

All of the walking machines described above use static stability. In contrast, the Carnegie-Mellon University Hopping Machine is the first successful machine utilizing active balance [15]. This machine is a one legged machine, and continuously hops in order to maintain its balance. It has a telescoping leg whose length can be changed. This action is powered by pressurized air from outside as a main
Figure 2.2: Logical decomposition of computational tasks in the ASV [14]
power source. It can also freely change its leg angle with respect to its body so that it can maintain its balance when it touches down on the ground, as well as move its body in the desired direction when it takes off from the ground. Specifically, the two leg angles are controlled by two hydraulic actuators.

2.3 Stepping Algorithms

It has been noticed for a long time that animals typically use their legs for moving their bodies in certain fixed periodic patterns known as gaits. Because of the lack of the technology needed to capture high-speed limb motions, scientific study of gaits was delayed until the invention of the motion picture camera. The first notable work in this area was performed by Muybridge [16] who developed a camera to take sequenced photographs and conducted experiments on several animals. From such photographs, he could see actual movements of legs while animals were moving, and was able to determine sequences of support by several legs repeated in a given manner.

The first work on gait analysis which utilized significant mathematical methods was performed by Hildebran [17]. His work provided a quantitative approach to the classification of symmetrical gaits using “gait formulas” which he devised. At about the same time, the modern theoretical approach was initiated by Tomovic [18], who assigned two states to each leg of an animal or a walking machine; one state is the supporting phase while a leg is on the ground and the other is the transfer phase while a leg is in the air. This finite state model has been further developed by McGhee and Frank [6] with additional mathematical descriptions and other refinements [5,6]. Adopting the idea of a sequential machine, McGhee defined a gait matrix $G$ which is a $k$-column matrix whose successive rows are binary $k$-tuples corresponding to the successive states of a particular gait of a $k$-legged
locomotion system. Therefore, the total number of the rows of the gait matrix is equal to the length of one cycle of the gait sequence [5], and entries of the gait matrix are either 0 (in support phase) or 1 (in transfer phase).

Later, McGhee and Jain, proposed another gait representation called the *event sequence* for a k-column gait matrix [19] (for a k legged walking machine). If the k legs are numbered 1 to k, then the event sequence number i refers to the touch down event of leg i, while the event sequence number i+k shows the lifting event of leg i. Therefore, the integers from 1 to 2k appear exactly once in the event sequence if legs are used in certain fixed periodic patterns. This event sequence is frequently used to describe gaits.

2.3.1 Stability

One of the most important characteristics of animals or legged vehicles is that they should maintain their balance at all times in order to stand or move. If this is not maintained by their legs or their body, they turn over. Basically, there are two possible methods to maintain balance.

The first method is to employ "dynamic balancing" [15,20] by carefully managing potential and kinetic energy. Though this method is usually adopted when a mammal or a man moves at high speed, it is complex because the stability is maintained with successive fall and recovery cycles while legs are placed on and lifted from the supporting surface [21]. Because living animals and human beings have extraordinary complexity in their muscle-powered degrees of freedom, only simplified dynamic models [22] have been employed for analyzing dynamic balancing. Until now, most studies of dynamic balancing have been concentrated on bipeds [21-25], and hopping machines having one, two, or four legs [15,20].

Though an animal or a man exhibits success in moving at high speed with
the dynamic balancing method, with few exceptions [15], current successful walking machines use a second method known as static stability because of the lack of computing power of current computers and transducing capability of current components. In order that a legged vehicle be statically stable, the projection of its center of gravity must be inside the polygon defined by its supporting legs on the supporting plane. Formally this polygon is called the support pattern, and its definition is as follows:

**Definition 1:** The support pattern associated with a given set of foot positions is the convex hull of the point set in a horizontal plane which contains the vertical projections of the feet of all supporting legs in the direction of gravity [6].

Therefore, if the vertical projection of the center of gravity of a vehicle is inside the support pattern, the vehicle is statically stable. McGhee and Frank [6] define the longitudinal stability margin which allows a degree of static stability to be calculated to permit comparison of various support patterns.

**Definition 2:** The instantaneous longitudinal stability margin at time t for an arbitrary support pattern is equal in magnitude to the shortest distance along the longitudinal direction from the vertical projection of the center of gravity of a locomotion machine to an edge of its support pattern [6].

Therefore, if a vehicle uses periodic leg movements for straight line locomotion in the direction of the longitudinal axis, then the longitudinal stability margin associated with a given gait pattern is the shortest instantaneous margin during one complete cycle. This stability margin concept is mainly useful for straight-line locomotion. If the legged vehicle is omni-directionally controlled, the definition
should be expanded [11,26]. Thus, the following stability definition is adopted for this dissertation:

**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 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.

Throughout the research work this dissertation, the stability margin defined in Definition 3 is used. However, it is recognized that this limits the algorithms to be developed in this dissertation to low speed motion.

### 2.3.2 Periodic Gaits

For an $n$-legged machine or animal, if every limb operates with the same cycle time, then the gait is said to be **periodic** [4]. Because such gait patterns are repeated periodically, they are easily parameterized by analyzing them in a certain leg cycle. Among several gait parameters, an important one is the **duty factor** $\beta_i$, which is defined as the fraction of a locomotion cycle during which leg $i$ is in contact with the supporting surface [4]. In general, each leg can have a distinct duty factor, but the class of gaits can be further limited by an expectation that if every leg of an animal or a machine were mechanically identical and evenly spaced along the body, then the task of supporting and propelling the animal or machine should be equally shared. In other words, this class of gaits has an identical duty factor for all legs. Such gaits are called **regular gaits** [5]. Thus, regular gaits can be represented by one duty factor. Regular gaits are generally adopted by animals when they walk on easy terrain [27-29]. Among the gaits in the regular gait class,
when a quadruped walks along a straight-line, McGhee and Frank [6] proved that the “quadruped crawl” optimizes the longitudinal stability margin. Specifically, when the range of the duty factor of the gait is from 3/4 to 1, the optimal gait is derived. Later, for hexapods, Bessonov and Umnov [30] found a class of optimally stable gaits for straight-line locomotion using the longitudinal stability margin, and the range of the duty factor of this gait class is from 1/2 to 1, which is exactly two times greater than that of quadrupeds. This class of gaits is known as wave gaits [28] because the leg stepping events on each side of the legged vehicle move from the rear to the front of the body during forward body motion, as if a wave propagated from the back to the front. This type of wave gaits is described more precisely as “forward wave gaits”, while in “backward wave gaits” the leg stepping events of the vehicle move from the front to the back of the body during forward body motion. Because wave gaits are a subset of the class of regular gaits the leg stepping events are governed by only one parameter, $\beta$, for a given vehicle geometry [31,32].

The longitudinal stability margin is increased either by increasing the duty factor [4] or by optimizing the leg stroke [32]. When this margin is applied for a vehicle to climb up a slope, the maximum slope is determined by this margin. Therefore, increased duty factor and decreased leg stroke are preferable in climbing up a slope. However, the maximum slope is determined by only the geometry of a vehicle [33-35].

Because of its predetermined optimal stepping sequence, if a wave gait is adopted for the locomotion of a legged vehicle, the only variation which can be applied to the leg motions is the stepping position within its kinematic limits at the beginning of its supporting phase. Thus, two variations have been proposed. The first one involves calculating a stepping position so that at the middle of the
support phase, the leg exactly passes the geometrical center of the area that the leg can reach on the supporting surface [33]. The other approach is to select a stepping position as far forward as possible without harming stability [34]. In order to realize this idea, the kinematic limits of the legs are restricted so that any supporting points on the ground included in the restricted kinematic limits can make the vehicle stable when one of them is selected as an actual supporting point. On a slope, the kinematic limits of the middle legs are dynamically changed according to the terrain slope to increase stability. All of these approaches are based upon the assumption that all portions of the terrain are equally accessible when selecting a foot touch-down position.

2.3.3 Free Gaits

If some portions of terrain present obstacles, such as holes, rocks, soft soil, and excessive slope, a totally different class of gaits, known as free gaits [10,11], is more suitable than periodic gaits, because this type of terrain allows neither a touch-down position which is calculated by predetermined methods without utilizing terrain conditions, nor a leg sequence which is set by preconditions, such as wave gaits. In free gaits, not only the leg sequence, but also the touch-down position of each leg is optimized in real time by consideration of the gait stability of a legged vehicle.

In what follows, first, two heuristic algorithms, which are related to this study, for free gaits for real time application are introduced. The first algorithm is proposed by McGhee and Iswandhi [10], and the second algorithm is one introduced earlier by the author [11], which is similar to that described in [10], although details differ. Specifically, both algorithms use the same type of terrain model, one which is composed of two types of terrain cells, either suitable for footholds (per-
mitted) or unsuitable (forbidden) [10,11]. Moreover, both algorithms utilize the same general strategy to maximize the number of legs in the air so that the vehicle can recover its stability by placing one of the legs which is presently in the air. However, when the stability test of the vehicle fails, the first algorithm immediately recovers the vehicle stability by stepping down an additional leg among the legs in the air, because the legs used by the first algorithm are weight-less legs so that they can be moved to the destination without spending any amount of time. Therefore, from this unrealistic assumption, it is to be expected that the legs are unnecessarily frequently changed, though this fact is not explicitly mentioned in the first work.

In order to overcome the above critical problem, two important concepts are implemented in the algorithm in [11]. The first is to introduce an eight state leg control sequential machine which simulates the physical characteristics of legs and allows a reasonable leg motion trajectory. The second is to make the algorithm predict into the future. If the algorithm checks only the present stability of the vehicle and finds that the vehicle is not stable, the vehicle will fall down because there is no time to recover the stability of the vehicle with legs with physical limitations. On the other hand, if the algorithm can foresee vehicle instability, then, using the legs in the air, it can prepare to make the vehicle stable before it actually becomes unstable. Thus, the prediction time period should be longer than the time needed to move a leg to support the body from a certain position in the air. For the actual implementation, the prediction time is chosen as the time required to step down a leg.

The second major difference of the two algorithms is that the first has only two body degrees of freedom that can be controlled by an operator, but the second has three degrees of freedom. Thus, the second algorithm can make the vehicle move
omni-directionally over terrain. Because of the additional degree of freedom in the second algorithm, two important definitions of the first work do not correctly work in the second work. These are the "kinematic margin" [10] and the longitudinal stability margin. In the second work, the first spatial margin is replaced with a temporal margin, and the second margin is replaced by the stability margin in Definition 3. Due to these new margin concepts, the roles of the kinematic margin and the stability margin are clearly distinguished in the second work. Thus, it can explicitly optimize the stability margin and the kinematic margin whenever it places a leg on the ground, while the first algorithm only deals with the kinematic margin to select a leg to be placed on the ground.

The last difference between the algorithms is the action which can be performed when the predicted state of the vehicle is not stable. The action performed by the first algorithm is simple leg selection and placement, while the second algorithm uses much more complicated reasoning to make the vehicle stable while finding a good supporting point for a leg, and even modifies the vehicle body trajectory to make the vehicle stable if the above support point selection cannot make the vehicle stable. It is observed that the total performance of the second free gait algorithm is greatly influenced by the leg placement method used by the foothold selection procedure for the legs in the air. However, the work of [10] does not explicitly state the foothold selection method.

Consequently, the work in [10] explored only the possibility of the use of the free gait with neither considering the physical leg constraints, nor explicitly trying to maximize the vehicle stability margin. Though both algorithms use the same general strategy, namely, that of minimizing the number of the legs on the ground, it is not apparent that the first algorithm minimizes the number of legs on the ground from its simulation output in [10], while no redundant legs are found in the
output of the simulation from the work in [11]. Even though the work in [10] is unrealistically simplified, it inspired the work in [11] as well as the research work of this dissertation.

The only other published approach to free gait implementation for hexapods is that developed by Patterson [36]. In this work, he devised a guidance algorithm which utilizes a deceleration plan whenever the vehicle is unable to comply with operator commands, and kinematic margins. Currently, his algorithm has been implemented and is being tested on the ASV. The research of this dissertation differs from that of Patterson in that it is intended for omni-directional control while in Patterson’s research it is assumed that the motion is mainly in a forward direction as in [10].

2.4 Programming Paradigms and Languages

The algorithm developed in this dissertation was not planned in a “top down” fashion and subsequently implemented, but rather it evolved gradually with continuous refinements because the definite goal and the requirements were not known precisely in advance, but continuously changed during the research effort. That is, the methodology used in this research was that of incremental refinement of prototypes. Provided that a programming environment with the proper characteristics is found, in such an approach the development cost will be minimized, as well as the cost of debugging, the cost of later changes, and the cost of communication to other persons. Moreover, a correctly chosen environment will force a programmer to implement the algorithm in a well organized way. However, no single programming paradigm is appropriate to handle all problems which can be met during the algorithm development. Like the various correct tools used by a carpenter, if possible, a chosen language or environment is expected to provide appropriate multiple
programming styles. In other words, different paradigms allow different things to be stated concisely, and provide different invariants under change [37]. This is the major concern of this section and also of Chapter 4. Therefore, the brief survey in this section is mainly intended as preparation for Chapter 4, and in Chapter 4, the actual comparison study is made to choose a well-suited environment and a programming language for this research work.

There are several programming paradigms, but in this section only the most common programming paradigms are described. These are *imperative*, *function oriented*, *object oriented*, and *logic oriented* [37]. First, conventional programming languages, such as Fortran and Pascal, mainly utilize the imperative programming styles. Thus, programs written in these languages tend to have large amounts of assignment statements and need to specify each control path through the programs in explicit detail. In other words, the programs are basically collections of mechanisms for routing control from one assignment statement to another [38]. The verbosity of this programming style tends to make the size of the programs large, thus in large and logically complex programs, the problem solving logic is seldom transparent. Moreover, because the meanings of the large number of variables are position-dependent in the program, the verification of the program is not an easy task. However, these programming languages are still very popular among scientific programmers.

LISP is a language which can utilize the functional programming paradigm [38,39]. In this programming style, functions can accept other functions as arguments, and return them as returned values, so that the programmer can define and manipulate functions and data with equal facility. In the LISP programming language, all data as well as all programs are represented by lists. LISP works with one simple basic mechanism, namely, application of a function to its arguments
Therefore, a list meaningful to be executed uses another list as its arguments, executes with them, and returns a value. The LISP programming language also facilitates manipulation of symbolic data because of its list processing primitives and the basic list data type. Using a lambda expression (an anonymous function) [38,39], LISP also manipulates functions like data. That is, functions defined by the lambda expression in LISP can be passed to a function as arguments or be returned as returned values. Though new control structures have been added into various LISP dialects, recursion is a basic control mechanism to all LISP implementations. Because of its simple data type and easily expandable characteristics, its application area is continuously growing, and several special-purpose LISP machines have been introduced. These machines not only improve the LISP programming environment, but also allow programmers to utilize other programming paradigms, like object-oriented programming, in addition to the basic LISP programming language.

In the object-oriented programming paradigm, objects are allowed to possess status and a set of behaviors which can be invoked by a corresponding message being sent from another object. Objects are “instantiated” as members of classes, and the objects from the same class have the same structure and the same behavior. The message passed to an object consists of a name for a behavior and optional arguments. The response to a message is determined by the object, not by the object class, because it is also affected by its object specific internal status. Because the objects are hierarchically related, the behaviors and status variables are inherited. Therefore, a programmer can easily define new class objects by adding new behaviors and new status variables to the previously defined class. Smalltalk is the first object-oriented programming language to receive widespread acceptance [40].
In the logic programming paradigm, the programming procedure is to declare the logical structure of problems, instead of explicitly specifying the detailed control flow to solve problems which are usually described by procedures in imperative style programming languages [38]. Thus, the computer carries out the details; i.e., how to accomplish the declared logical problems. Therefore, nonprocedural programming is possible with this programming style. Prolog is the major programming language with this paradigm. Programs in Prolog are expressions in the form of propositions that assert the existence of the desired result, and Prolog tries to satisfy its goals with “backward” reasoning [38].

2.5 Summary

This chapter provides a survey of previous work relevant to the problems considered in this dissertation. First, a brief historical review of important walking machines developed during the last twenty year period is presented. Among them, the ASV used for the model vehicle of this study is described in some detail including operational modes, but its geometrical dimensions are not included in this chapter. Rather, this information will be provided in Chapter 3.

In Section 2.3, walking machine stability is discussed, including three definitions related to stability. Following this topic, two types of gaits are introduced. The first type is periodic gaits, in which gait patterns are repeated periodically. Practically, wave gaits, a special case of the periodic gaits, are frequently implemented on walking machines because of their simplicity and optimal stability. Thus, two stepping algorithms which utilize wave gaits are briefly described. In the last subsection, free gaits are discussed based on a comparison between two similar, but different in detail, free gait stepping algorithms. Finally, a simplified free gait stepping algorithm which utilizes the kinematic margin and vehicle
deceleration is described.

In the last section, four programming paradigms are described along with example programming languages for each. In Chapter 4, these languages will be again compared in order to select one language which will be used to implement the algorithm developed in this dissertation study which utilizes omni-directional control of body motion and free (non-periodic) gaits over terrain containing regions not suitable for footholds. The following chapter will present detailed problem statement of this study.
CHAPTER 3

Detailed Problem Statement

3.1 Introduction

The development of an obstacle avoidance algorithm which utilizes a free gait algorithm is the major objective of the research presented in this dissertation. Throughout this research work, the Adaptive Suspension Vehicle (ASV) is used as the vehicle model. Therefore, this algorithm is implemented on and tested using the geometry of the ASV. For simulation purposes, prismatic terrain is devised, and its surface is simplified as a binary surface which has two types of regions: permitted (safe) and forbidden (unsafe).

In this chapter, a kinematic model of the vehicle adopted for the simulation study is presented, and the terrain model is introduced. After presenting the vehicle model and the terrain model, sensor models are discussed which can be utilized when this free gait algorithm is implemented in the ASV. Finally, operator interactions with this algorithm are described.

3.2 Vehicle Model

Because the vehicle model adopted in this dissertation is the ASV, the important dimensions of this vehicle model are the same those of the physical ASV, but minor portions are modified for simplicity. That is, the cab and the terrain scanner are omitted from the simulation vehicle model, while the other important
geometries of the body and the legs are identical to that of ASV. Consequently, the vehicle body of this simulation is represented by a simple six-faced box, and the legs are drawn as two line segments as shown in Fig. 3.1. The six legs of the ASV are named Leg 1 through Leg 6, with the odd numbered legs attached to the left side of the vehicle body with respect to the forward body longitudinal axis, and the even numbered legs are to the right side of the body.

3.2.1 Body Model

As shown in Fig. 3.1 and Fig. 3.2, the top of the body is wider than the bottom in order to provide adduction capability for the legs. The three legs of each side of the body are almost evenly spaced, and the knee direction of the back legs is opposite to that of the front and middle legs.

The origin of the body coordinate system is located inside the body. If the origin is observed from the top of the body, it is seen at the geometrical center of the top view of the body. On the other hand, if the origin is observed from one side of the body, it is seen 1.08 feet down from the top plane of the body as shown in Fig. 3.2. The $x$ axis of the body coordinate system is parallel to the body longitudinal axis and is directed forward, while the $z$ axis of the system is directed upward from the origin. Because the body coordinate system is a right-handed system, the $y$ axis points toward the left side of the body when it is observed from the top of the body.

From this body coordinate system, all the body and leg positions can be described, but frequently it is necessary that the same points should be measured in the earth coordinate system. For this purpose, a $4 \times 4$ homogeneous transformation matrix, $H$ [41], is defined, which relates body coordinates to earth coordinates. Therefore, the $H$ matrix can be used to transform the position described in the
Figure 3.1: Kinematic simulation model of the Adaptive Suspension Vehicle
Figure 3.2: Body geometry of the model vehicle
former coordinates into the latter coordinates. Specifically, a given point vector expressed in body coordinates can be transformed to earth coordinates using the following relation [41]:

\[
[X_E Y_E Z_E]^T = H[X_B Y_B Z_B]^T
\]  

(3.1)

where the subscript \( E \) means earth coordinates, and the subscript \( B \) denotes body coordinates. The two point vectors of both sides of the above equation correspond to the same point in three-dimensional space.

The matrix \( H \) is also interpreted as a matrix which contains four different pieces of information regarding body coordinates with respect to earth coordinates. From this fact, the \( H \) matrix can be decomposed into four submatrices: a \( 3 \times 3 \) matrix \( R \), a \( 3 \times 1 \) matrix \( p \), a \( 1 \times 3 \) matrix \( t \), and a \( 1 \times 1 \) matrix containing the value 1 [42]. The decomposition is

\[
H = \begin{bmatrix}
R & : & p \\
\ldots & \ldots & \ldots \\
t & : & 1
\end{bmatrix}
\]  

(3.2)

The first submatrix, \( R \), shows the rotational relation of body coordinates with respect to earth coordinates. If the inside of the \( R \) matrix is examined, then the following characteristics can be observed: the first column of the \( R \) matrix shows the vector representation of the x axis of body coordinates with respect to earth coordinates, and the second column and the third column show the relation of the y axis and the z axis to earth coordinates, respectively. The second matrix, \( p \), represents the amount of translation from earth coordinates to body coordinates. Based on the most general use of the \( H \) matrix in computer graphics, the lower left 1x3 submatrix, \( t \), represents perspective transformation, and the lower right, 1, is the global scaling factor. In this study, the \( t \) matrix is always set to \([0 \ 0 \ 0]\), and
the scaling factor is fixed to 1 for simplicity. Because of the characteristics of the $H$ matrix, incremental body movement respect to earth coordinates can be easily computed if the incremental relation is known in body coordinates. The relation between a newly updated $H$ and an $OldH$ becomes

$$NewH = OldH + OldH^H \Delta$$

(3.3)

where $^H \Delta$ is an incremental body movement with respect to body coordinates. Therefore, $OldH^H \Delta$ can be considered as incremental body movement with respect to earth coordinates [41].

3.2.2 Leg Model

In order to simplify leg kinematics, a pantograph mechanism has been adopted for the ASV legs [12]. Due to the characteristics of this mechanism, the foot position of the leg is determined by the displacement of its lift and drive axes in its base plate shown in Fig. 3.3 with the fixed ratio, 5 in drive, and 4 in lift. Therefore, its kinematic limits are determined by the geometry of its base plate. In other words, its foot can reach any point in a subset of a planar two-dimensional space which is limited by its base plate dimensions. Consequently, two degrees of the ASV leg movement freedom in three-dimensional space are realized by this mechanism. The other degree of freedom of the ASV leg movement is associated with the hinge which connects the leg to the vehicle body. The axis of this hinge joint is parallel to the direction of the forward axis of the vehicle body (or parallel to the x axis of the body coordinates). Because this axis is perpendicular to the above two axes, the foot of any leg can reach points in three-dimensional space. Specifically, the hinge joint allows the leg to swing in and out about the body x axis (adduction/abduction). Therefore, the leg can reach points in a three-dimensional
Figure 3.3: Geometry of the base plate and its foot position
space defined as a fraction of a torus whose cross-section is determined by the geometry of the structure of the pantograph. This space is called the working volume.

**Definition 4:** A *working volume* is associated with each vehicle 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 [11,34].

**Kinematics of the ASV Leg**

Before describing the detailed dimensions of the working volume of the ASV leg, its kinematics will be described in this section. Using the pantograph mechanism described in the above section, the kinematics of the leg can be easily derived. If the foot position of Leg 1 is called \([X_F Y_F Z_F]T\) in body coordinates, then its kinematic relation can be written as

\[
\begin{bmatrix}
X_F \\
Y_F \\
Z_F \\
\end{bmatrix} = \begin{bmatrix}
5d_m + X_{hip} \\
(4d_L + L) \sin \theta + m \cos \theta + Y_{hip} \\
-(4d_L + L) \cos \theta + m \sin \theta \\
\end{bmatrix} \quad (3.4)
\]

where \(\theta\), \(d_M\), and \(d_L\) are the joint variables. The variables \(X_{hip}\) and \(Y_{hip}\) denote the appropriate hip position in the body coordinate system. Since the hips of all the legs are attached to the body in the plane containing the x and y axes of the body coordinates, there is no displacement in the z axis direction for the hip of Leg 1, while there exists \(X_{hip}\) and \(Y_{hip}\) displacements along the x and y axes. This equation can be applied to the other five legs if the signs and the hip position values are appropriately modified because all the legs have the same basic geometry as that of Leg 1.
The knee position $[X_K Y_K Z_K]^T$ of Leg 1 can be also easily obtained. The position is

$$
\begin{bmatrix}
X_K \\
Y_K \\
Z_K
\end{bmatrix}
= 
\begin{bmatrix}
S_3 \cos \alpha + X_{hip} \\
(S_3 \sin \alpha - (d_L - L)) \sin \theta + m \cos \theta + Y_{hip} \\
-(S_3 \sin \alpha - (d_L - L)) \cos \theta - m \sin \theta
\end{bmatrix}
$$

(3.5)

where

$$
\alpha = \frac{\pi}{2} - \tan^{-1}(d_M/d_L) - \cos^{-1}\{(S_1^2 - S_2^2 + d_L^2 + d_M^2)/(2S_1 \sqrt{d_L^2 + d_M^2})\}
$$

(3.6)

As for the foot position of Leg 1, this relation can be also applied to the other five legs by changing signs and the $x, y$ hip position values appropriately.

**Working Volume of a Leg**

Because of the pantograph structure of the legs, their kinematic limits are determined by the mechanical limits of their base plate shown in Fig. 3.3. But in order to eliminate interference between the middle legs and the front legs or the back legs, the bounds on the drive axis variables of the legs, $d_M$, are slightly reduced from their mechanical limits. The joint limits used for this simulation study are

$$
-20^0 < \theta < 20^0 \quad \text{for Legs 1, 2, 3, 4, 5, 6}
$$

$$
0.5833' < d_L < 1.4167' \quad \text{for Legs 1, 2, 3, 4, 5, 6}
$$

(3.7)

$$
-0.35' < d_M < 0.5833' \quad \text{for Legs 1, 2, 5, 6}
$$

$$
-0.4417' < d_M < 0.4583' \quad \text{for Legs 3, 4.}
$$

Since the drive axis is totally independent from the hinge axis, the foot movement along the longitudinal axis of the body is not influenced by the joint variable,
\( \theta \). Before investigating the effects of two joint variables, \( \theta \) and \( d_L \), the foot coordinates are temporarily moved from body coordinates to the hip of one of the legs in order to simplify the kinematic relations. Then Eq. (3.4) is changed to

\[
\begin{bmatrix}
X_F \\
y_F \\
Z_F
\end{bmatrix} = \begin{bmatrix}
5d_m \\
a \sin \theta + m \cos \theta \\
-a \cos \theta + m \sin \theta
\end{bmatrix}
\]

(3.8)

where \( a = 4d_L + L \) and \( m = 1.0467' \) which is the offset of the leg from its hip socket. From Eq. (3.8), it can be observed that \( Y_F \) and \( Z_F \) are related by the variable, \( \theta \). Therefore, the foot movement can be expressed as

\[
Y_F^2 + Z_F^2 = a^2 + m^2 = R^2
\]

(3.9)

where \( a = 4d_L + L \). If \( a \) is fixed (or \( d_L \) is fixed), then the foot can draw a portion of a circle limited by the joint variable, \( \theta \), while the radius, \( R \), can vary from 3.316' to 6.564' by changing the value of \( d_L \). Therefore, the cross-section of the working volume in the \( Y_F - Z_F \) plane is a segment of an annulus as shown in Fig. 3.4. If this area is extended in the body longitudinal axis using the joint limit variable of \( d_M \) in Eq. (3.6), then the working volume of one of the six legs is derived, which is shown in Fig. 3.5.

The free gait algorithm of this dissertation very frequently uses the working volumes associated with the legs in order to determine the usefulness of the legs. In order to speed up the computations related to the working volumes, their shapes should be simpler than the original working volume. This further simplification is called the constrained working volume and is shown in Fig. 3.6. The upper and lower portion of the working volume are replaced with planes which are parallel to the X-Y plane of the body coordinate system. The cross-section of the working volume is a symmetrical trapezoid, but it is not symmetrical to the z axis of its

34
Figure 3.4: Cross-section of working volume in $Y_F - Z_F$ plane
Figure 3.5: Working volume of ASV leg

<table>
<thead>
<tr>
<th></th>
<th>Legs1,2,5,6</th>
<th>Legs3,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xmax</td>
<td>2.9165</td>
<td>2.2915</td>
</tr>
<tr>
<td>Xmin</td>
<td>-1.75</td>
<td>-2.2085</td>
</tr>
</tbody>
</table>

Units: ft
Figure 3.6: Constrained working volume of ASV leg

<table>
<thead>
<tr>
<th></th>
<th>Legs 1, 2, 5, 6</th>
<th>Legs 3, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xmax</td>
<td>2.9165</td>
<td>2.2915</td>
</tr>
<tr>
<td>Xmin</td>
<td>-1.75</td>
<td>-2.2085</td>
</tr>
</tbody>
</table>

Units: ft
hip socket, because the leg is slightly offset to the outside from the body by the
amount of m. The dimensions of the constrained working volume devised for
this simulation study are shown in Fig. 3.6, but its length in the x axis of body
coordinates is dependent on the leg number.

3.3 Terrain Model

When a walking vehicle walks on terrain, a proper terrain representation
should be utilized in order to plan its motions using the free gait algorithm studied in this dissertation. This terrain representation can be derived from terrain
information gathered by sensors, such as the ASV terrain scanner[14]. In this
study, among various possible representations, a simple binary type terrain model
is adopted, which is composed of two types of terrain cells. One type of a terrain
cell is called a permitted cell, which is suitable to support the body load when a
leg steps on the cell, and the other type is named a forbidden cell, because it is
not suitable to support the body load due to unfavorable terrain conditions in the
terrain cell, such as excessive slope, soft soil, holes and rocks, etc. Though this
terrain model is simple, the terrain classification from actual terrain data obtained
by sensors is still subject to further study, and it is not included in this disserta-
tion. Therefore, manually or randomly generated terrain models are used for this
study instead of those produced from physically existing terrain.

A typical terrain utilized in this study is shown in Fig. 3.7. This is a prismatic
terrain [34] which is a function of x only. The z axis of the earth coordinate system
is defined in the opposite direction of that of terrestrial gravity. This type of terrain
is relatively simple to generate as well as adequate to test the terrain adaptation
of a free gait algorithm. In this program, the height information is given only by
a human operator, but the terrain classification information is input either by an
Figure 3.7: Typical simulation terrain with the model vehicle
operator or the automatic obstacle generation program which randomly produces obstacles using randomly generated numbers and a ratio between permitted cells and forbidden cells given by a human operator [11]. When the terrain is manually classified by an operator, he can designate the terrain cell which is under the cursor to either type permitted or forbidden, and position the terrain cursor on any cell in the model terrain. The terrain cells classified into forbidden cells are graphically marked with a cross so that they are distinguished from the other type of cells on the model terrain during the simulation.

The dimensions of the terrain cell are one foot by one foot in the x and y direction of the earth coordinate system, respectively, and these dimensions are not changed regardless the slopes of the cells. In other words, though the terrain cells have different values in the z axis of earth coordinates, if the cells are projected on a flat horizontal imaginary plane, then all the cells have exactly the same dimensions. Therefore, a cell area on the model terrain surface will be different from those of the cells which have not the same slope of the given cell.

The dimensions of the terrain sample are 39 feet by 39 feet in the x and y direction of the earth coordinate system, and one whole model terrain contains 1521 cells because the cell dimensions are one foot by one foot. The terrain classification information is internally saved in a two-dimensional array for simplicity, because one cell is matched one-to-one with one element of the array. Moreover, the location information of one of the cells is easily computed from the index of the array. Because of the prismatic terrain model, the height information is saved in a separate one-dimensional array containing 39 z-axis values. Again, the position information is computed from the index of the array.
3.4 Sensor Model

The free gait algorithm of this dissertation utilizes terrain information classified into two different types of cells described in the previous section. In order to gather terrain data being used to classify terrain, an optical terrain scanner can be used, which is mounted on the top of the ASV cab located at the front end of the vehicle. This scanner is a 3D sensor using a GaAlAs laser diode which sweeps over the terrain with an azimuth angle coverage of approximately ±40 degrees from the ASV longitudinal axis, and an elevation angle coverage of -15 degrees from the horizon to -75 degrees by mechanical movements of mirrors[14]. This mechanical scanning is repeated at a 2 Hz rate, and the returned frame data from the scanner is a 128 x 128 array of 8-bit range values. In order to measure the range from the scanner to the terrain, a sinusoidally modulated laser light beam is transmitted and reflected back to the vehicle. The phase of the returned signal modulation envelope is compared with the transmitted signal and range information is generated from the phase difference between the two signals. Because the modulating frequency is 15.36 MHz, the ambiguity interval, which can be considered as the maximum range of detection, is equal to 32 feet [14,43].

Besides the above scanner, the vehicle has additional sensors to provides its inertial reference information. They are three linear accelerometers, three rotational-rate sensors, and a two-axis vertical gyro [14]. Various information from the above sensors helps the vehicle to maintain the body position and orientation based on the terrain information obtained from the terrain scanner.

In order to control the ASV legs, the vehicle monitors three tachometers, three potentiometers, and three pressure sensors of each of the legs. These sensors are connected to one of the six leg computers, and the data from the sensors are
utilized to determined velocity, position, and force information for each leg.

3.5 Human Interface

Generally speaking, the free gait algorithm studied in this dissertation performs real-time optimization on leg sequencing and stepping positions using desired body positions. Because the vehicle body is modeled as a rigid body which has six degrees of freedom in three-dimensional space, in order to control a multi-legged vehicle using this algorithm, a human operator provides proper information for the vehicle so that desired body positions can be determined. This information can be one of following types: positions, velocities, or accelerations. In this study, velocity information is given to the vehicle for the above purpose [44].

If each one of the six degrees of freedom is respectively controlled by one velocity component, then six different velocities in body coordinates are necessary to control the vehicle. But it is not a simple task for a human operator to simultaneously control six different velocity components. Therefore, the number of control inputs is reduced with the aid of a body regulation control scheme. This scheme, which will be described in detail in Chapter 5, automatically generates three velocities using terrain data. The generated velocities are one linear velocity (z-directional velocity in body coordinates) and two rotational velocities (rotational velocities around the x and y axes of body coordinates). Therefore, the remaining three velocities are commanded by a human operator. These velocities from an operator control can be decomposed into the following motions: forward-backward motion, side-to-side motion, and turning motion around the z axis of the body. These motions can be freely mixed to control the vehicle, but their magnitudes are limited. The maximal magnitudes of each velocity component are shown in Table 3.1.
Table 1:
Maximal magnitudes of each velocity component commanded by an operator

<table>
<thead>
<tr>
<th>Component</th>
<th>Magnitude</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward/backward</td>
<td>2</td>
<td>ft/second</td>
</tr>
<tr>
<td>Side-to-side</td>
<td>1</td>
<td>ft/second</td>
</tr>
<tr>
<td>Turn-in-place</td>
<td>$1 \times 10^{-1}$</td>
<td>radian/second</td>
</tr>
</tbody>
</table>
3.6 Summary

This chapter presents important information relating to this dissertation, such as the vehicle model, the terrain model, the sensor models, and the nature of human interaction with the vehicle control system. In order to produce practical results, the existing ASV is selected as the vehicle model because the selected model geometries can influence the performance of the free gait algorithm developed in this dissertation. Naturally, it is expected that the performance of this algorithm on the physical ASV can be predicted to same considerable degree from the results of this simulation study.

In addition to the vehicle model, the terrain model also plays an important role in free gait algorithm performance. For practical purposes, in this dissertation, the terrain is idealized and simplified as prismatic terrain which has a binary-cell type surface. The terrain surface contains two terrain cell groups, such that one group contains permitted cell regions, and the other group forbidden cell regions. Because of this simplification, searching and evaluating terrain along a vehicle trajectory which is designated by a human operator command is possible with reasonable computations. Moreover, it is expected that this type of terrain data can be produced by the terrain scanner mounted on the ASV eventually because it can resolve elements comparable in size to the terrain cells used in this study.

The number of velocity component commands given by a human operator is limited to three in order to simplify the task of controlling the vehicle. The remaining three velocity components, which are necessary to complete the six velocity components required to specify the exact motion of the vehicle body in three-dimensional space, are generated by the body control scheme which utilizes terrain data. Basically, the three velocity components generated by the scheme
help the mobility of the vehicle on rough terrain. A detailed discussion of this topic will be given in Chapter 5.

The following Chapters 4 and 5 will present leg and body controls, and in parts of Chapter 6, foothold selection and body deceleration methods will be discussed. Based on the discussion in Chapters 4, 5, and part of Chapter 6, the free gait motion coordination algorithm is represented with hierarchically arranged flow charts at the end of Chapter 6. Chapter 7 includes results and evaluations of the free gait algorithm.
CHAPTER 4
Control of Individual Legs

4.1 Introduction

Because of the non-periodic characteristics of free gaits, leg cycling time is difficult to predict or unpredictable, while for periodic gaits, leg cycling time is easily determined from pre-defined leg sequencing. Thus, in periodic gaits, leg movement can be controlled as a synchronous event sequence for each leg. On the other hand, in a free gait, leg cycling cannot be governed solely by a synchronous mechanism, but must also include asynchronous mechanisms which can handle the uncertainties of a free gait in leg sequencing. In this study, the above two different mechanisms are implemented as synchronous and asynchronous leg control states respectively.

In this chapter, a leg trajectory and associated controlling states for a free gait algorithm will be described. Then the leg velocity control state block of the overall leg coordination algorithm developed in this dissertation will be coded in several languages. Specifically, Pascal, LISP, Prolog, OPS5, and Smalltalk will used for this purpose. The programs written in these languages are listed in Appendix A of this dissertation. The chapter concludes with a comparison of results and a decision regarding language selection for the rest of this dissertation.
4.2 Leg Modeling

4.2.1 Leg Trajectory

Choosing a form for the basic leg trajectory is arbitrary to some extent. One possible trajectory, used in this study, is shown in Fig. 4.1. The trajectory of Fig. 4.1 is drawn with respect to the body coordinate system, and the direction of body movement is indicated by an arrow which is directed to the right side of the page. As can be seen, one complete trajectory cycle is divided into seven distinct segments, corresponding to seven foot velocity control states: Ready, Advance, Descent, Contact, Support, Lift, and Return. Foot behavior in each of these states is described in the following paragraphs.

In the Ready state, the foot position is stationary with respect to the body, and remains stationary until the Deploy command is given to the leg by the free gait algorithm. When the leg is in the Ready state, the foot position should be high enough to clear obstacles on the ground in order to eliminate the possibility of hitting them with legs in the Ready position. Because of this reason, the value of the z direction of the foot position in the body coordinate system is variable depending on terrain conditions, but fixed during the Ready state for any given leg cycle in order to simplify planning and to eliminate unnecessary energy consumption. Thus, the z value of foot position is decided by the higher level planner when the Return state is started. In this dissertation, at the end of the Lift state, the leg is lifted 1.4 feet from the ground in order to clear terrain obstacles of up to this height. Therefore, if this height is maintained in the Return state as well as in the Ready state, the vehicle can avoid the possibility of hitting obstacles with the leg. This idea is actually implemented in the following way: The foot ground clearance information measured in earth coordinates is translated into a corresponding value
Figure 4.1: Leg motion relative to body during forward body motion over level terrain
along the z axis of the body coordinate system. This translated value is used as the z value of the position of the Ready state as well as that of the Return state. Therefore, as shown in Fig. 4.1, the z values of both states are the same in the body coordinates.

The vehicle in this study can be controlled omni-directionally, and consequently, stepping positions can be at any position inside of the area that the leg can access at a given time. For handling this uncertainty, the horizontal coordinates of the Ready position are set to the center point of the reachable area which can be accessed when the vehicle is parallel to a flat level terrain with the origin of the body coordinate system located 5.4 feet above the ground.

In the Advance state, a leg moves from its position in the Ready state and stops at the position which is directly above the desired future supporting point. The distance from the desired point on the terrain is set as 1.4 feet for the reasons described above. The trajectory of this state is a straight line with respect to the body coordinates for simplicity, and the leg movement of this state is completed in 0.6 seconds. In this state, the value of the z axis in the body coordinates may be varying during leg movement, but on flat level terrain this value will remain unchanged as shown in Fig. 4.1. That is, on uneven terrain, foot movement during the Advance state may be either up or down relative to the body in order to ensure a ground clearance of 1.4 feet at the start of the Descent state.

In the Descent state, the foot is lowered in the direction of gravity while tracking a desired future supporting point until the foot touches down on the terrain. Thus, the leg trajectory is a vertical line with respect to earth coordinates in order to eliminate the possibility of striking obstacles with the side of the leg during descent. For planning purpose, the nominal travel time for this motion is internally set to 0.4 seconds, but practically, slight deviations from the nominal
time may be expected because of inaccuracies in measurement or mechanical errors in leg movement. To simulate this situation, a small error in ground position is intentionally introduced into the simulation of leg descent control. If the distance between a foot position and the terrain is less than some small amount, then, within the simulation program, the leg is considered as having contacted the terrain. In this dissertation, the amount of the error in estimation of foot elevation above the terrain is fixed as 0.02 feet. This number was arbitrarily chosen merely to ensure that actual foot touch-down does not occur at the planned time. Real errors of this type in the physical ASV are likely to be larger.

During the Contact state, while the foot is on the terrain, the leg takes up its prescribed portion of the body weight [45], and starts to support the body. The duration of this state can be set to any reasonable time, but one second is chosen for the simulation study because one second is more than a sufficient amount of time for settling of the reaction force between the terrain and one of the ASV legs. The other purpose of the time constraint is to prevent too frequent leg changes. Without this constraint, there is a possibility that the free gait algorithm can make a leg lift as soon as it touches down on the terrain. This action should be avoided for practical reasons, such as mechanical limitations and energy consumption. The trajectory of the foot position in this period, if the movement is observed from earth coordinates, will be stationary unless the foot is slipping on the terrain. If the foot is seen from body coordinates, its position is moves in an exactly opposite direction to the body movement. Specifically, the foot velocity of leg i relative the body motion, $V_{Fi}$, is given by [34]

$$V_{Fi} = -V - \omega \times BPF_i$$

(4.1)

where $V$ is the translational body velocity, $\omega$ is the rotational body velocity, and
\( B P_{F_i} \) is the foot position of \( \text{Leg}_i \). The four vectors in Eq. 4.1 are expressed in the body coordinate system.

In the \textit{Support} state, a leg supports and propels the body until the free gait algorithm decides to remove it from a support pattern by issuing a \textit{Recover\_command}. The foot movement is exactly the same as in \textit{Contact} state so long as there is no slippage with respect to the terrain.

During the \textit{Lift} state, a leg moves in a direction exactly opposite to that of the \textit{Descent} state. At the end of this state, the leg is elevated from the terrain by 1.4 feet. This movement is achieved in a nominal time of 0.4 seconds, which is within the capabilities of the physical ASV leg. However, there is no reason why the foot should accurately reach the imaginary point which is in the air and somewhat arbitrarily located. Therefore, controlling a leg in the \textit{Lift} state is simpler than in the \textit{Descent state}. A slight deviation is acceptable, and this state can be terminated by a timing event. Consequently, the time-out increment for this state is taken as 0.4 seconds regardless of whether the leg has not yet reached or has already passed the desired position.

In the \textit{Return} state, the leg returns to the position of the \textit{Ready} state, which is fixed in the body coordinates. This state lasts during a time interval of 0.6 seconds which is an arbitrary value, but which represents a reasonable amount of time for the physical ASV leg to finish the desired movement. For simplicity, the trajectory of this state is chosen to be a straight line in body coordinates while maintaining the value of the foot \( z \) axis position as described in the discussion of the \textit{Ready} state.

51
4.2.2 Foot Velocity Control State Transitions

Using the characteristics of leg motion discussed in the previous section, one seven-state sequential machine is needed for the control of each ASV leg. The state diagram of this machine is shown in Fig. 4.2. This machine can control a leg by sending a velocity command to a leg, while using the actual leg position at a given time and terrain height as feedback information. Except for the Ready state, all six states output non-zero velocity commands in the body coordinate system. Among the seven states, four states are synchronous states and the other three states are asynchronous states. Therefore, four timeout constants and three terminating events are necessary for the synchronous and asynchronous states respectively. While the four timeout constants can be simply labelled as T1, T2, T3, and T4, the three terminating events can be categorized into two groups by the level at which they are generated in the control software hierarchy of the free gait algorithm. In the first group, there is only one event, Contact-confirm, which is sent from a lower level than the foot velocity control level, when a leg touches down on the terrain. This event may be monitored by hardware in the form of a contact sensor or a proximity sensor. It will generally occur around 0.4 seconds after the start of the Descent state (this amount of time is used for planning the stepping action). The other group contains two commands, Deploy-command and Recover-command. These commands are generated by leg motion planning software which resides at a higher level than the foot velocity control block. Differing from the transition from the Descent state to the Contact state, these commands can be generated at any time by the leg motion planning block, because no pre-specified time is associated for planning this transition. Consequently, if the time constants associated with the synchronous states and the planned nominal time of 0.4 seconds for leg descent
Figure 4.2: Foot velocity control state diagram for free gait algorithm. (All states command leg rates using actual leg position feedback.)

T1: 0.6 seconds
T2: 1 second
T3: 0.4 seconds
T4: 0.6 seconds
are taken into account, the shortest leg cycle time will be about three seconds.

For making comparisons among several computer programming languages, the foot velocity control program is slightly modified so that it can run independently. Thus, the terminating commands and events are simulated inside one piece of code. One simple way to simulate the two commands and one event is by making them available whenever any asynchronous state tests terminating commands or an event. Because of this implementation, as soon as the leg state goes into an asynchronous state, it is ready to change to the next state, but because of the sampling effect, which allows only one state transition at a time, any asynchronous state will make a transition to another state only after one sampling time. Therefore, any asynchronous state lasts only one sampling time in the modified leg control program.

4.3 Pascal Implementation

The Pascal language is chosen as the example language to represent the imperative style of programming. Pascal was chosen for this purpose because it enjoys a growing popularity among scientific programmers and also because most parts of the ASV control program are written in Pascal [46,47].

From the discussion of the preceding Section 4.2, two commands and one event are generated inside of the leg control program. As listed in the Pascal program of Appendix A, the two commands and one event are realized with set type variables: namely, leg_command, and leg_event. These variables are assigned all possible cases as soon as the program is started. While the program is running, condition checking of an asynchronous state is done by testing membership on one of the two set type variables which represent the commands or the event respectively. For example, because the Ready state is one of the asynchronous states, the program
tests whether deploy\_command is a member of leg\_command. Clearly, this test is always passed. Thus, leg\_state is changed to advance and leg\_clock is reset. This implementation is purely arbitrary, but was chosen for programming convenience.

In addition to the above variables, to improve program readability, the full state names are used for the actual code. That is, leg\_state is realized by scalar type [46,47], which defines an ordered set of values by enumerating identifiers which denote the values.

For continuing iteration until \textit{end\_time}, the \textit{repeat} function built into Pascal is utilized. The structure of this function starts with \textit{repeat}, is followed by a piece of code, and ends with \textit{until}. The termination condition of the function is followed by an \textit{until} statement, and if the condition is met, then the iteration is terminated. In this specific program, \textit{end\_time} \leq \textit{time} is the termination condition checked at the end of every iteration. Therefore, if \textit{time} reaches the \textit{end\_time}, this program is terminated.

The code inside the \textit{repeat} loop is divided into two functional blocks. The first block simulates the foot velocity control state diagram, while the second block simply prints out current information after converting a state name to a readable form. This is needed because a scalar type constant cannot be directly written on a computer terminal screen. In order to execute a correct piece of code depending on a foot velocity control state name, the \textit{case} conditional statement is employed. The \textit{case} structure starts with a \textit{case} statement, a piece of code, and ends with an \textit{end} statement. When the case statement receives a state name through leg\_state, which is the argument variable of the case structures following the \textit{case} statement, the portion of code following the name of the state is executed.
4.4 LISP Implementation

As the name LISP suggests (List Processing), all components of a LISP program are symbolic expressions, such as lists or atoms [38,39]. Lists in LISP have zero, one, or more than one items inside a pair of parentheses. Each item can be a list or an atom, and an atom can be a number or a symbol. When a list is evaluated, the first item of a list is considered as the name of a function, and the rest of the list is recognized as arguments of the function. After evaluation, a value is returned. The returned value is again a list or an atom depending on the evaluated function. Because, in LISP syntax, data and procedures use the same list type, virtually any list can be evaluated, but a value is returned only when a meaningful list is processed. If a simple data list is evaluated, then an error will be reported. Therefore, LISP code can generate other executable code because of the above mentioned characteristics. In turn, LISP is capable of treating equally any lists whether they represent data or functions as arguments or returned values of a LISP function if they are meaningful to the function.

In interpretive mode, evaluation is performed by a part of the LISP interpreter which continuously repeats a read-eval-print loop. All three parts of the loop can be separately accessed by a human user if he wants to use only one part of the loop. Read and print functions convert human readable forms of lists to internal forms and back again, respectively. The eval part evaluates a list read by the read function, and returns it to the user through the print function.

The LISP implementation of the foot velocity control finite state machine is listed in Appendix A. For the purpose of fair comparison with the use of Pascal, the same number of variables and the same variable names are used in the LISP program. One exception is the data type representing the two commands and
one event. These variables are implemented by lists instead of sets. Not only are list type variables natural choices in LISP, but also set type variables are hard to implement in LISP. Specifically, for initializing the variables used in the program, the \textit{setf} built in function of LISP is utilized. Though the basic functionality of the \textit{setf} function is to assign a value to a variable, this function can accomplish multiple assignments. If the \textit{setf} in a \textit{setf} list is excluded in counting the items in the list, odd items of the list are variables and even items are values to be assigned to their corresponding variables. For example, in the first \textit{setf} list, $t_1$, $t_2$, $t_3$, $t_4$, and \textit{state} are assigned with the appropriate values of the even items in the list.

The overall structure of the LISP implementation is remarkably similar to that of the Pascal implementation. One of the advantages of LISP over Pascal for programming foot velocity control is the ease with which symbols can be manipulated such that natural state names can be directly utilized in the program without special attention when the names of states are printed on the screen. In order to output the name of the current foot velocity control state, in the Pascal implementation, the case statement is used, but in the LISP implementation, no similar structure is needed beyond adjusting the output format.

The iteration and conditional structures used in LISP have a different syntax from Pascal, but the basic functionalities are very close each other except that LISP offers more flexibility. In particular, the iteration function, \textit{do}, of LISP has three functional blocks. In the first block following the function name \textit{do}, local variables are assigned new values before starting a new iteration. This multiple assignment can be done sequentially or in parallel depending on an option * which can be added to the \textit{do} function. That is, $do*$ performs sequential assignments, while $do$ performs parallel assignments. Because $do$ has no * in this program, the two local variables, \textit{time} and \textit{leg_clock}, in the \textit{do} function are assigned simultaneously, and
are incremented by 0.1 second per loop, while they are initialized as zero using the middle values of their lists when the do function is initiated. The second block of a do is a conditional block, which checks a terminating condition before executing its body (the third block). Though more than one list can be in a conditional block, only the first list of the block is tested. When the test succeeds, looping is terminated and, in addition, the series of operations following the first list of the block is performed as side effects and the result of the final operation is then returned as the value of the do function. In the program, 'done is the only operation following the testing function. Because ' prohibits evaluation in LISP, the returned value will simply be "done".

The conditional function, cond, is one of two parts of the third main functional block of the do function in the LISP program of Appendix A. Like the case statement in Pascal, the cond function can make possible a multi-way branch. Each branch is sequentially tested from the top until a condition is satisfied. Basically, each branch is a list, and the first item of each list is usually evaluated to test whether the branch will be selected or not. If a returned value from the first list is not nil, then the rest of the lists of the branch are evaluated one by one. As in the second block of a do function, only the last value is returned by the cond function though several lists may be evaluated. If only one list or one symbol is in one branch, then that list or symbol is used for both testing and returning a value. If the value evaluated is not nil, then the branch is selected and the evaluated value is returned as a returned value. In the program, the last branch of cond is simply (T). Because T is always evaluated as T, i.e., not nil, if all the previous branches are evaluated as nil, then T is returned as a returned value of the cond function. Thus, no change on the foot velocity control state is made.

The other part of the third block of the do function consists of printing rou-
tines. Differing from the Pascal program, it can print out state names without any special conversion from the internal form of the name of the state to the printable form on the screen. For making easily readable output, two separate lists, \textit{show-time} and \textit{show-state}, are built before printing out time and state name.

4.5 Prolog Implementation

Prolog is an example programming language for investigation of logic programming languages, and is programmed with facts and rules inside the Prolog environment \[48\]. For dealing with the leg control state problem, a practical difference between Prolog and the two previous languages is whether or not there exists a capability to assign values to variables. In Prolog, a variable is instantiated not by an assignment but by a matching. Variables are locally instantiated in one clause unless they are passed as arguments to other clauses. That is, if locally instantiated variables are stored as facts of Prolog, such information can be preserved globally in Prolog because facts are saved in the global database of Prolog. In Prolog syntax, a fact is written in two parts, a relationship, and object(s), terminated by a period. Because Prolog is case-sensitive, the relationship of the fact should be started with a lower case letter. The object(s) following the relationship are enclosed in a pair of parentheses, and more than one object can be delimited by comma(s). Objects may be either a constant, such as a name, or a variable expressing any universal object. The former should start with a lower case letter, but the latter with a upper case letter. Besides facts, logical relationships can be expressed with rules in Prolog. That is, \textit{if then} logical relations can be expressed by using the Prolog reserved symbol, \texttt{:-}. The right side of \texttt{:-} corresponds to the \textit{if} part, and the left side of \texttt{:-} is the \textit{then} part. The \textit{if} part of a Prolog expression (clause) can have one or more conditions which can be AND-ed or OR-ed
depending on delimiters between the conditions. If a comma is used as a delimiter between two conditions, they are AND-ed, but if a semi-colon is used, then two conditions are OR-ed. In Prolog, each condition is considered as a new goal, called a "subgoal" in the following text. Differing from the if part, the then part (rule header) has one item. This is considered as a goal, and succeeds only when all the subgoals in the right side of the rule are satisfied.

As can be seen from the listing of in the Prolog program of Appendix A, all time constants, leg states, commands, and an event are asserted as facts at the beginning portion of the program. For a direct comparison of the program written in Prolog with the program written in LISP, commands and events are realized with facts containing lists. As discussed in Section 4.2, the implementation in Prolog also provides a true condition whenever asynchronous states test commands or an event for their state transitions. Another special feature of the Prolog program is the means used to create a loop. To realize a loop, the backtracking facility built in to Prolog is utilized.

Basically, Prolog always makes an attempt to satisfy a goal by matching a fact (or a rule header) to a goal (or a query). When a matching fact (or a rule header) is found, Prolog proceeds to the next goal (or the subgoals in the rule) while, in the database of the Prolog environment, Prolog marks the place where the matching occurred, and instantiates any variables that have matched. Specifically, Prolog scans its database which contains facts and rules from top to bottom, and from left to right inside a rule. If it cannot find a matching fact (or rule head), it attempts to re-satisfy the previous subgoal. This attempt continues until no subgoal can be re-satisfied. If no subgoal can be re-satisfied, then Prolog discards the rule and attempts to find an alternative rule to satisfy the goal by uninstantiating any variables which became instantiated when the previous rule was matched. Prolog
resumes scanning its database from the previously matched rule but not including
the previously tried rule. This mechanism is known as backtracking [48].

In this program, the built-in predicate repeat is used to implement a loop. This
predicate provides a way to generate multiple solutions through backtracking; i.e.,
by making the goal, repeat, always succeed again on backtracking. Therefore,
repeat causes a repetition of the cycle, time(T), T >= Final_time again and again
until no backtracking is done to the repeat when the three subgoals are satisfied.
Only when T is greater or equal to the Final_time given, the three subgoals are
satisfied. Thus, the leg(Final_time) clause is succeeds, and the program is stopped
because no more backtracking occurs.

In addition to the predicate repeat, another predicate, !, (the cut symbol) is
used. Cut is another way to affect the backtracking mechanism. Simply stated, cut
acts like a wall in the flow of logic satisfaction. Therefore, cut prevents Prolog from
considering cycle again during backtracking. Specifically, when Prolog encounters
a cut symbol, it will erase the database pointer which is used when backtracking
is started. Thus, if cut symbols are placed in proper positions in a program, they
will speed up its execution and save memory because Prolog does not have to
keep track of the backtracking pointer beyond cut symbols. In the program under
consideration, the cut symbol at the end of the cycle clause is essential to prevent
undesired backtracking through the cycle rule when backtracking is transferred
from the time(T) subgoal. Thus, when backtracking is started by the last subgoal,
T >= Final_time, the cycle subgoal is skipped, and then the left subgoal, the
repeat, of the cycle subgoal is tried. Because repeat always succeeds, cycle is
started again, entering from the left-hand of the cycle rule.

At the beginning of the Prolog program, as in the previous examples, time
constants are initialized by asserting predicates t1, t2, t3, t4, and dt. The time and
leq.clk predicates keep track of timing events of the program, and the state predicate stores the name of the current state. Leg commands and the Contact.confirm event are asserted in order to simulate them in the way discussed in Section 4.2.2.

This program is started by issuing a query, the leq predicate with a desired amount of time inside the parentheses. When this query is terminated by a period and is followed by a carriage return, Prolog starts to pick subgoals of the leq rule one by one from left to right, and tries to make them succeed. Thus, Prolog repeats the loop until \( T \) reaches the final.time in the way described in the previous paragraph.

Because update.time(\( T_l \)) in the cycle rule always succeeds, \( T_l \) is always updated with a new time. This mechanism works like a procedure of an imperative programming language because the special feature of Prolog, backtracking, is not utilized. On the other hand, the new.state(\( Y \)) subgoal in the cycle rule uses the backtracking facility to make a correct state transition. Prolog tries to match the new.state(\( Y \)) subgoal of the cycle rule with the first clause in the data base, new.state(advance). If this is the first loop encountered when the program is started, \( Y \) is instantiated with advance because the present state is ready and then the next subgoal, command([deploy,\( \cdot \])), always succeeds. But, in the second loop, because the present state is not ready, the first new.state clause fails. The backtracking facility of Prolog then tries the next new.state clause. Though the state(advance) subgoal succeeds in the second clause in the data base, the final subgoal, \( T \geq T_l \), does fail. Therefore, another backtracking is initiated, and the third clause is tried. This try will also fail, and the rest of them will continuously fail except for the last clause. The last one always succeeds, because no special condition is included in the right side of the rule except for matching of the right-hand side, state(\( X \)). Thus, \( X \) of the last new.state is instantiated with the name of
the present state. This rule will be used repeatedly except when the foot velocity control state is changed.

4.6 OPS5 Implementation

The OPS5 language is used for writing forward-chaining rule-based programs for artificial intelligence (AI) applications. It shares the general characteristics of a pure production system which is composed of rules, a data base, and an interpreter [49]. In OPS5, these components are respectively realized in production memory, working memory, and a recognize-act cycle [50,51].

The WM (Working Memory) of OPS5 is composed of ordered pairs, a time tag and a working memory element. The time tag associated with the working memory element is a numerical identifier supplied by the OPS5 interpreter, and is utilized for resolving conflicts among rules. This method of conflict resolution always succeeds because the same time tag is never assigned to more than one working memory element. The working memory elements which represent objects or facts of the real world can be simple numbers, symbolic atoms, or structures containing attributes and values. These elements also can be created, modified, or destroyed by a user.

The production memory of OPS5 is filled with productions (or rules). One rule consists of a LHS (left-hand side) and a RHS (right-hand side). The LHS is composed of patterns called terms, which may be matched with working memory elements kept in WM. If a PS (production system) rule is recognized as an if and a then part [52], the LHS is comparable to the if part, and RHS is the then part. Therefore, if the condition of a rule is satisfied (if all the condition terms of the LHS are matched with working memory elements in WM), then its RHS is executed. The RHS contains an unconditional sequence of commands;
i.e., actions on WM, such as creating, modifying, or destroying working memory elements. Specifically, OPS5 provides seven action types: change working memory, manipulate files, output information, assign values to variables, call user-written subroutines, make the interpreter stop firing productions, and add production rules to production memory [50,51]. Syntactically, a rule is enclosed in one pair of parentheses, and starts with the symbol $p$. The name of the rule, the LHS of the rule, the symbol $\rightarrow$, and the RHS of the rule follows the symbol $p$.

The recognize-act cycle of OPS5 basically follows the conventional recognize-act cycle which is sequentially executed the three parts; i.e., match, rule conflict resolution, and act. Unless a halt action is performed, these three parts are executed again and again. If a halt action is performed, or no LHS of rules are satisfied, then control is returned to the user. But the cycle of OPS5 is slightly modified from this conventional recognize-act cycle. Specifically, OPS5 checks the halt action after the match step instead of after the act step. This modified cycle insures that the conflict set, which is the collection of all rules with the condition terms of their LHS matched with WM elements, is consistent with the current contents of working memory when the cycle is ended by a halt action.

The OPS5 version of the program for the foot velocity control state block is listed in Appendix A. This program is composed of a variable type declaration part and ten rules. The first rule initializes all the variables declared in the first part of this program, and the rest of the rules use the initialized variables in order to update states as well as to update global time and leg time. One of restrictions of pure PS's like OPS5 is that direct communication between production rules other than the data base in WM is not allowed. A rule in OPS5 can effect other rules only by modifying the data base in WM so that other rules may act from the contents of WM left as the trace of the previously fired rule.
In OPS5, only one rule can be fired at one time, but the program controlling foot velocity should perform state transitions as well as increasing and checking global time and leg time, in order to properly terminate the program. This latter action occurs when the time reaches the desired amount of time input through the \textit{final.time} variable provided by the user when the program is started. If each rule includes both the time increment and the state transition functions, this scheme will work, but will cause unnecessary repetitions for all the rules. For correcting this inefficiency, the common functional pieces that increase and check global time and leg time are factored out, and made into a separate rule while additional tags are introduced. This rule is the second rule in the program listed in Appendix A, and the two tags explicitly control firing sequences of rules in production memory so that a rule fired can select which type of a rule will be fired in the next cycle; i.e., one cycle is devoted to increase the two time variables and to check whether the global time has reached the final time or not, and the next cycle is assigned to perform a state transition based on an event or commands or on the increased leg time by the previous rule. After a rule which can make a state transition is fired, it forces the next cycle again to do the time updating and checking operations by changing the tags. Therefore, one of these two types of operation is executed every other cycle. In this program, the flag \textit{(action change.time)} makes OPS5 do time updating and checking operations, while the flag \textit{(action change.state)} makes OPS5 do a state transition. When the second rule is fired by the OPS5 interpreter, the tag \textit{(action change.state)} is set in WM so that the next rule to be fired should be one of the rules which may change \textit{leg.state} or the last rule which does not change \textit{leg.state}. At the end of one of these rules, the tag \textit{(action change.state)} is removed from WM and the tag \textit{(action change.time)} is set in WM so that the second rule, \textit{increase.time}, can be fired in the next cycle. Specifically, in order to
exchange flags in WM, two OPS5 operations, *make* and *remove*, are utilized. The former operation is functionally similar to the *asserta* predicate of Prolog, and the latter operation is analogous to the *retract* predicate of Prolog. These functions will be described in detail in a later paragraph of this section of this dissertation.

In order to start the OPS5 program in Appendix A, one seed must be inserted into the working memory of OPS5 because OPS5 cannot start without any working memory element matched with any LHS of some rule. In this program, this seed is input by a user before starting the program. Specifically, *start 1* is the seed which fires the first rule which initializes all the variables and accepts a user input from the terminal to set a value for the *final_time* variable. Before finishing the actions of this rule, an "Initialization is done message" is printed on the screen.

All the variables used in this program are realized with the structured data type of OPS5. Because such structures should be declared before being used in a program, a list of structures of variables are declared at the beginning of the OPS5 program in Appendix A. The declaration is started with *literalize*, and is followed by a class name and a list of attributes which will be associated with their corresponding values. For example, the class name of the first structure is *time_constant*, and its attributes are *t1, t2, t3, t4*, and *dt*. Those attributes will be assigned proper values in the initialization rule described in the previous paragraph, and can be accessed with the prefix operator ` in front of a name of an attribute. In the first rule, the value of *t1* is initialized to 0.6 by ` t1 0.6.` Though an attribute may have one scalar value at a time, a *vector-attribute* type allows one attribute to have more than one value because OPS5 allows the number of values assigned to attributes declared as the *vector-attribute* type to grow or to shrink dynamically. Leg commands and leg event attributes are declared as a *vector-attributes* for this purpose.

66
Basically, the LHS of a rule tests conditions by matching with elements in WM. Because the LHS can contain more than one term, only when all of these terms are matched with the corresponding parts of the WM elements is the RHS of the rule executed. Syntactically, each term is enclosed with a pair of parentheses, and the first element of each term is a class name of a structure. If the class name is found in WM, then a search is made for the attribute of the term through the element associated with the class in WM. If the attribute of the term is found, and the value of the attribute is equal to the value in WM, then the matching process succeeds. If the value of the attribute of the LHS term is a variable, then this matching always succeeds, and in addition the variable is instantiated with the value of WM. For example, in order to match the second term of the LHS of the second rule with elements in WM, OPS5 searches its WM to find an element whose class name is \textit{time}. After this element is located in WM, OPS5 checks the attribute \(^t \) in the element found. In the OPS5 program of Appendix A, these two attempts will always succeed. After the attribute is found in the element, OPS5 tries to match the value of the attribute of the term in the LHS and that of the element in WM. In this case, because the value of the attribute of the term is a variable \(<time>\), this try to match always succeeds, and \(<time>\) is instantiated with the value of the matched element in WM. (Syntactically, a variable of OPS5 is enclosed with "<" and ">".) The OPS5 interpreter will repeat this process with the next term of the LHS. If the next term is successfully matched, then OPS5 repeats this process until all the terms are tested. Thus, all the conditions expressed in the terms of the LHS of one rule are logically ANDed because of the matching method used by OPS5. If any one of them is not matched with the elements in WM, OPS5 skips the rest of the terms of the LHS, and starts to test the LHS of the next rule. During this matching process, if the value of an attribute is numeric, then it can
be compared numerically, such as greater than, or equal.

The RHS of a rule is composed of one or more actions which is enclosed by a pair of parentheses, and each action can have one or more operations. In the OPS5 program of Appendix A, only four different types of operations are used by the RHS of the rules. The first type is make, which creates an element in WM. Syntactically, make is followed by a class name, and a set of attribute and value pairs. The second type is the remove operation. Remove is followed by one number, and this number represents a pointer to the element in WM which is matched in the LHS of the rule; e.g., (remove 1) means to remove the WM element matched by the first term of the LHS of the rule. In the second rule, (remove 1) deletes (action change_time) from WM. The third type is the modify operation. This operation can change one or more values of an element in WM. Equivalently, two operations, remove and make, are performed by one modify operation. Like the remove operation, this operation also has a pointer represented by a number. The first action of the third rule modifies the value of the state attribute from ready to advance. The number 2 in this action points to the element whose name is leg.state in WM because it is matched by the second term of the LHS of the third rule. The last action type is the compute operation. Compute allows five numerical operations: addition, subtraction, multiplication, division, and modulus. This operation is used in the first and the second actions in the second rule to update the time and the leg.clock.

4.7 Smalltalk Implementation

Smalltalk is one of the best known examples of an object-oriented programming language. Programs in such languages execute by sending messages to objects [40]. An object represents one component of a Smalltalk system, and may correspond to an object in the real world. For example, an object of Smalltalk can be
a number, a character, a window in the Smalltalk system, or even a leg object simulating a physical ASV leg. Each object possesses its own methods regarding how to provide a behavior when it receives a message. For example, one of the objects representing leg states in the Smalltalk program of Appendix A knows how to make a state transition to the next state when it receives a state transition message. An actual state transition of this object will be performed when its internal condition is matched with its precondition for a state transition in addition to the state transition message. Specifically, the precondition may be a time or an event depending on the type of the state, either a synchronous or an asynchronous state respectively, and this type of information is provided to the leg state objects by the leg object class when the program is started because the leg object class owns several leg state objects in this program. But in a Smalltalk environment, because the preconditions for terminating foot velocity control states are internally stored, the information about preconditions can not be directly accessible, but can be known to the outside only by sending a proper message. Specifically, a Smalltalk object has private storage to save its current status, and a set of methods allowing communication to another objects. A Smalltalk object makes visible its abstracted characteristics to the other objects in the Smalltalk system by hiding its detailed implementation from the others except for a small number of data communication channels. These are important concepts relating to "module independence" [53].

An object in a Smalltalk system can be considered as an independent module. For example, the foot velocity control state diagram of the ASV has seven different states. The details of those states are buried inside of the state objects of Smalltalk, and they respond to the same message, doTransition. Therefore, in order to make a state transition, it is enough to send the doTransition message regardless of the types of state objects. Consequently, in a Smalltalk system, in order to utilize
objects, it is sufficient to know what kind of messages the objects can respond to instead of how the objects perform given operations.

Though all the seven states have their own unique names in order to be distinguished from each other, they can be grouped into two different types, asynchronous and synchronous. In this Smalltalk program, the seven states are represented by seven distinctive Smalltalk objects that have their own unique names, but state transition methods can be described depending on the state type, either synchronous or asynchronous. Therefore, depending on the types of the states, the seven state objects can be produced by two different prototype states which are abstract versions of the seven state objects, while each prototype state represents either an asynchronous state or a synchronous state. In a Smalltalk system, the prototype states can be expressed as two distinct classes, and seven state objects can be created as instances of either class. In other words, a Smalltalk class defines common characteristics of the instances of the class by describing a set of methods, a set of messages that may be executed by corresponding methods, and instance variables (or private storage) for storing object specific characteristics or status. Therefore, objects instantiated by the same class in Smalltalk have common characteristics (what kind of messages they can understand and how to behave on receipt of given messages), while the objects can be distinguished by instance variables that can store unique information.

Another aspect of Smalltalk is the concept of subclasses which allows hierarchical classification and inheritance among classes. For example, both synchronous states and asynchronous states of the leg control state diagram have common factors, such as names of themselves and pointers to next states. These two common characteristics may be factored out to build an abstracted state which cannot stand alone except by adding more pieces of definition to make either of the two
types of states. If a timeout constant is added to their common characteristics as a trigger to make a state transition, this turns out to be a synchronous state, while if a termination event is added, an asynchronous state is derived from the above abstracted state. If the logic flow is reversed from the above discussion, then it can be said that the synchronous and the asynchronous states inherit the common characteristics from the abstracted state. The abstracted state can be implemented as a class in Smalltalk, and then the synchronous state class and the asynchronous state class will be defined as subclasses which inherit the common characteristics from the abstracted state class, named \textit{State} in the Smalltalk foot velocity control state program.

Basically, programming in Smalltalk involves creation of classes by defining a set of methods defining behavior upon reception of a message from another object and instance variables for private storage. In this program, the \textit{State} class has two instance variables, \textit{name} and \textit{nextState}. Syntactically, in Smalltalk, an instance variable should be started with a lower case character in order to represent the instance variable as a private variable which will be known only inside of an object instantiated by the class, while name of a class begins with a capital character in order that the name of the class is globally known to any objects in a Smalltalk system. In addition to the instance variables, the \textit{State} class has four methods, and one of them will be executed on reception of a corresponding message which is basically the name of the method. For example, \textit{getName}, one of the four methods of the Leg class, returns the name of the state of an object, and is activated by the \textit{getName} message. These four methods really act like communication channels to other objects, and only through the methods can another object retrieve the contents of \textit{name} or \textit{nextState} instance variables. This is a key feature to provide modularity of information storage because detailed implementation is hidden so
that later changes in this class does not have a global effect, but instead involves only isolated changes inside the modified class.

The Smalltalk program in Appendix A is composed of five class definitions: State, SyncState, AsyncState, Leg, and LegExecutor. The SyncState and AsyncState classes inherit two instance variables and four methods from the State class. By adding two instance variables (timeOut and time), and six methods to the State class, the SyncState class is devised, which will be used to make objects possessing the characteristics of a synchronous state. On the other hand, by adding one instance variable, event, and four methods to the State class, the AsyncState class is derived, which will instantiate objects possessing the properties of an asynchronous state. Differing from an asynchronous state, a synchronous state is terminated by a temporal event. Therefore, the SyncState class has two instance variables, timeOut and time, which keep track of the internal status of a synchronous state. Consequently, six methods are added to the State class because two methods are necessary to handle each instance variable, and two methods are used to initialize and make transitions of synchronous state objects.

The Leg class has a totally different hierarchy from the above three classes because there is no logical hierarchy between a leg and states, but the Leg class simply uses state objects. In this implementation, the state objects are locally confined as instance variables inside an instance of the Leg class. For example, if more than one leg object is in the Smalltalk environment, then localized information should be saved separately for each leg. A synchronous state should keep track of a timing event, but it should not be mixed up with the time of another synchronous state which has the same state name but is associated with another leg. Thus, each leg object needs its own state objects. One of the instance variables of the Leg class, legState, actually points to the current state of a leg object,
and an operation is performed when a proper message is given to this variable. Besides the above mentioned instance variables of the Leg class, two additional instance variables are defined, events and samplingTime. The latter variable simply stores the sampling time interval, 0.1 second, and the former variable simulates an outside event or commands to terminate asynchronous states. This variable always provides a true condition whenever a terminal condition is checked by an asynchronous state. In this implementation, the events variable is instantiated by the Dictionary class supplied by a Smalltalk package. The Content of an object instantiated by the Dictionary class is accessed by a key using the message at:. In order to store a value at the location pointed to by a key, a put: message is used. In this program, the key is the name of the state, and its corresponding content is either a command or a terminating event. Both of them are started with # in order to show that it is a symbolic constant.

The Leg class has three methods: changeState, getState, and initialize. The first method causes a state change depending on the type of a current leg state and its internal status, if a proper external event is provided as an argument. The second method simply returns the current state of a leg object. In order to find the name of a state, a getName message should be given the returned state by the second method. In the LegExecutor class, the same procedure is used in order to get the name of a state before printing it out on the terminal screen while the program is running. The last method initializes all of the instance variables, such as time constant, events and states. One special feature of the last method is the use of local variables named t1, t2, t3, and t4, which are only known inside of the last method. Local variables of this method are enclosed in a pair of — symbols, written at the beginning of the method.

LegExecutor, the final class of this program, gives life to a leg by sending a
message, changeState for each loop until globalTime has reached endTime. The value for endTime is given through the method argument which is followed by the endTime message and colon in the first method of the LegExecutor class. The iteration structure is similar to a conventional structure such as the while structure of Pascal, but the structure starts with an iteration condition statement inside square brackets and is followed by whileTrue: (iteration statement) and a body inside square brackets. The body updates globalTime by 0.1 seconds, sends a changeState message, and prints the name of a state on the Transcript window. To do this, the getState message is given to the aLeg object input through the argument of the execute method of the LegExecutor class, and then getName is given to the returned state from the previous operation. After this operation, the printOn message is sent to the name of the state. Because the name of the state knows how to print its contents on the screen, the name is printed on the screen. In this program, the state name is printed inside the Transcript window which is given as the argument of the message, printOn.

4.8 Summary

In dealing with the problem of controlling one leg, this chapter compares five programming languages. Code size is the easiest program characteristic to be compared. The program written in LISP has the shortest code size, and the programs in Prolog and Pascal occupy second and third places, respectively. The sizes of the programs in these three languages are almost the same in length, but the other two languages, OPS5 and Smalltalk lead to much longer code sizes. Among these two, OPS5 has a slightly shorter code size. Execution speed comparison is not easy because all five languages are not supported on one computer. Furthermore, because Pascal and LISP compilers are often available, these two languages can
exhibit substantially increased execution speed when their codes are compiled. On the other hand, compilers are not generally supported for Smalltalk, OPS5, and Prolog. Therefore, programs written in these languages generally run slower than those in Pascal and LISP.

The execution speed of each language is highly dependent on the computer hardware on which the language is implemented. For example, if programs written in the LISP language are compiled on a special LISP machine, its execution speed can be almost as fast as that of a Pascal program running on a comparable-sized minicomputer. If a special Prolog machine becomes available, (such as the Japanese Fifth Generation computer system [54] which can execute compiled Prolog programs in parallel) Prolog would run much faster than a Prolog program running with a Prolog interpreter implementation on a general purpose computer.

If the computer languages under consideration are compared for ease of making a loop, Pascal, LISP, Smalltalk can be said to have essentially the same capability. Because of the recognize-act cycle of OPS5, a loop can be made for controlling a leg, but executing simultaneously two tasks in one cycle is somewhat awkward. Thus, the two tasks which can be done inside one loop by the above three languages are executed in two OPS5 cycles. For making a loop in Prolog, a special facility, backtracking, is utilized together with the repeat predicate.

If the program is expanded beyond controlling a leg, or if the controlling method is changed, the modularity of a programming language plays an important role. Because Smalltalk is built on modularity concepts with hierarchy (an object-oriented programming style), Smalltalk exhibits the highest modularity among the five languages. In a certain sense, OPS5 can be considered as possessing almost the same characteristics with regard to modularity as Smalltalk, because each rule can be added or deleted without interfering with the other rules. But differing from
Smalltalk, OPS5 has no hierarchy among the rules. The other languages, LISP, Prolog, and Pascal, can be programmed in a modular style. Among them, LISP can be programmed very close to Smalltalk using recursion, Lambda expressions, and extensions of the LISP language which are not used in this sample program, but which are used in the free gait program of the following Chapter 6. The extensions are *structures* and *flavors*, which allow an object-oriented programming methodology within a Lisp environment [55]. It is significant that Zeta LISP supports non-hierarchical programming (multiple inheritance) which is not supported by Smalltalk [55,56].

In addition to the above considerations, the possible types of hardware available for execution of the selected language in real time and in the environment in which the ASV will be tested are considered to be important considerations in selecting the language to be used in the remainder of this dissertation. Since all applications programs in the current version of the vehicle control software are written in Pascal [14], this presents a strong argument for using this language. On the other hand, since a Symbolics LISP machine is also available to this project, it may prove to be feasible to connect such a computer to the existing computer either by a cable or by a radio link. Furthermore, it is expected that military LISP machines suitable for embedded applications will be available at an acceptable cost in about two years. Since the execution of LISP code on such machines should be as efficient as the execution of Pascal on a computer employing a conventional architecture, this is an argument in favor of the use of LISP in the following chapters. At present, a similar argument cannot be made for any of the other languages considered.

Yet another consideration in choosing a language for the implementation of the free gait algorithm of this dissertation is the ease with which programs can
be developed, debugged, and explained to others. In the author's MS thesis [11],
a preliminary version of a free gait algorithm was written in Pascal. To test the
relative ease of LISP programming, this program was rewritten in Common LISP.
It was found that the code size was reduced by more than a factor of two and
that the resulting program was much easier to understand, modify, and explain to
others. Moreover, the incremental compilation capability associated with the LISP
programming environment (VAX Common LISP) made debugging substantially
easier.

While the above remarks are admittedly somewhat subjective, the author's
preference for LISP was reinforced when a Symbolics 3640 LISP machine became
available to him during the course of writing this chapter. With the LISP machine,
three programming styles, the imperative, function-oriented, and object-oriented
paradigms, can be easily utilized from the LISP programming language without
complicated interface problems among them. The greatly enhanced programming
environment provided by this machine when taken together with all of the other
factors discussed above leads to the selection of Common LISP as the language of
choice for the remainder of the work of this dissertation.
5.1 Introduction

Because the ASV has six degrees of freedom in space, six velocity component commands can be used for controlling its body. Among the six velocity components, three are sent from outside the free gait program by a human operator or by an autonomous navigation system. As discussed in Chapter 3, these three components are composed of two translational velocities in the x and y axes of the body coordinates, and one rotational velocity around the z axis of the body coordinate system. Though the magnitudes of these velocities are limited to the values described in Chapter 3, the rate of change of velocity was not limited. However, rapidly changing velocities can degrade the performance of the free gait algorithm of this thesis because the algorithm uses events which are predicted one second into the future by using the present input command velocities. Therefore, smoothing the changes in velocity commands is important for increasing the performance of the free gait algorithm. For this purpose, velocity commands are filtered by a first-order filter before being utilized by the algorithm in order to plan and to execute the body motion and the leg motions. The remaining three velocities, one translational velocity in the z direction and two rotational velocities about the x and y directions with respect to the body coordinate system, are automatically generated by a body regulation method which utilizes terrain data so that
the vehicle can traverse rough terrain smoothly [34]. Because raw terrain data is too noisy to be used as input data for body regulation, before being used by the regulation method, the raw terrain data are smoothed by a least squares planar regression estimation method [33].

5.2 Command Velocity Regulation

This scheme used for filtering of joystick commands is shown in Fig. 5.1. In this figure, only one component is illustrated among three input command velocities. The quantity $v_c$ is one of input command velocities and the quantity $v_o$ is the output of this scheme which will be fed into the free gait algorithm so that the algorithm can plan motions.

Depending on the feedback gain of the scheme shown in Fig. 5.1, the performance of the free gait program can be improved, and the psychological effect on a human operator can be changed because the gain governs the behavior of the system. Because this system realizes a first-order low-pass filter, a rapidly changing velocity command can be considered as a high frequency input to this system. Therefore, a rapidly changing input velocity command can be filtered out, and the output of the scheme is mainly affected by the low frequency components of the velocity command. The feedback gain is the reciprocal of the time constant of the corresponding first-order low-pass filter.

In the system shown in Fig. 5.1, if the gain is too small (if the time constant is too large), then the output which is used by the free gait algorithm for planning purposes remains almost constant. The slowly changing output velocities can reduce the projection error, but the vehicle does not follow the desired input velocity command. In this case, for the human operator, the vehicle seems uncontrollable, because of the delayed response. If the psychological effect on a human operator is
Figure 5.1: Command velocity regulation scheme
of little importance, this latency problem would seem to be not critical to vehicle control. But the free gait algorithm itself uses this scheme to stop or to slow down the vehicle in order to avoid conditions such that the vehicle becomes unstable when the complexity of terrain is increased. Thus, a time constant which is longer than the planner prediction time, one second, may not be desirable for this system because the latency problem will not allow for the vehicle effectively to stop before being unstable. Clearly, a very small time constant is not suitable for smoothing purposes, either. Therefore, for this study, the time constant is set to 0.5 seconds (or the gain is 2), so that the algorithm can make the vehicle stop in a reasonable amount of time, if necessary. A detailed representation of a method to decelerate or to stop the vehicle motion will be presented in the next chapter. In the implementation presented in this thesis, the calculation of the velocity smoothing scheme shown in Fig. 5.1 is performed numerically, with an integration interval of 0.1 second, which is the same as the duration of the free gait computational cycle.

Though the above scheme is able to smooth the changes of an input command velocity, the acceleration of the vehicle is not quantitatively limited. For safe operation of the physical ASV, it is desirable that limiting of acceleration be accomplished. For this purpose, a limiter is inserted in front of the integrator of the command velocity regulator. The magnitude of this limiter is set as one tenth of gravitational acceleration or $3.2174 \text{ feet/sec}^2$. Therefore, in any direction commanded by a human operator, the acceleration of the vehicle will not exceed the maximum value, one-tenth of gravitational acceleration.
5.3 Terrain Adaptation

5.3.1 Automatic Body Regulation

The main object of controlling attitude and altitude of the vehicle body is to help increase the static stability and mobility of the vehicle. On flat level terrain, controlling the attitude and altitude of the body is trivial. Obviously, making the body parallel to the terrain is the best choice among various possible ways to control the body attitude in such circumstances. If this attitude control strategy is utilized, then the vehicle can obtain maximum stability from a given set of supporting legs, and the altitude of the body can be adjusted so that the vehicle can have a maximum reachable area for each leg. If this idea is adopted, then the origin of the body coordinates may be elevated 5.7313 feet from the ground as discussed in Chapter 3. In the actual free gait program, this height is slightly reduced to 5.4 feet for safety reasons because the reachable area of the ASV leg is rapidly reduced beyond the height which can give a maximum reachable area. But the value, 5.4 feet, is somewhat arbitrarily chosen, and may be altered if necessary. 

On uneven terrain with slope, the control of posture involves more complexity than for the flat level terrain case in order to attain the goal mentioned in the previous paragraph. An approach devised by Lee [34] is adopted in this study. Generally speaking in Lee's approach, if the slope of the terrain is increasing, the body attitude is inclined toward the direction of the slope of the terrain. That is, the body is rotated from a horizontal position to partially, but not fully, compensate for the terrain slope. This tends to increase the vehicle stability because the area of the vehicle body which is projected in the direction of gravity on an imaginary horizontal plane is increased beyond what it would be for full slope compensation. This larger area can give a better chance to produce greater stability.
with the given supporting legs. The reduced body altitude also helps the vehicle to have a larger degree of stability because the center of gravity is effectively shifted in the opposite direction of the projections of the terrain normal on the imaginary horizontal terrain, and the shifted amount is linearly proportional to the vehicle altitude. Therefore, to achieve the main goal of body control, the attitude and the altitude of the vehicle body is adjusted as a function of the slope of a given terrain.

Specifically, in this study, the following control parameters are used in order to control attitude and altitude of the vehicle body. The body altitude is linearly decreased from 5.4 feet to 4.4 feet when the angle of the slope is increased from zero degrees to 30 degrees. But the body inclination angle is linearly increased from zero to 10 degrees while the slope of the terrain is increased from zero to 30 degrees. Thus, on flat horizontal terrain, the height measured from the ground to the origin of the body coordinates is 5.4 feet, and the inclination angle is zero so that the body is parallel to the terrain as shown in Fig. 5.2. On the other hand, if the angle of the terrain slope is increased to 30 degrees, then the height of the origin of body coordinates is decreased to 4.4 feet from the terrain plane, and the inclination angle of the body becomes 10 degrees as shown in Fig. 5.2. That is, the body plane assumes an angle of 20 degrees with respect to an imaginary horizontal plane. In between these two extremes, the height and the inclination angle of the body are linearly changed as a function of the angle between the imaginary horizontal plane and the terrain plane. These relationships are shown in Fig. 5.3. Beyond the maximum angle, 30 degrees, the body height and the inclination angle are saturated, and do not decrease or increase any more. Operation on slopes beyond 30 degrees is not expected because such operation would be dangerous or even unstable, but a terrain clearance of 0.13 feet with 13.25 feet body length is
Body control on a horizontal terrain

Body control on a terrain with 30 degree slope

Figure 5.2: Two extremes of body control
Body altitude control strategy

Body inclined angle control strategy

Figure 5.3: Body altitude and attitude control strategy
nevertheless maintained even at this extreme angle [35].

Up to this point, the terrain has been treated as a flat plane so that it can be represented with one angle variable. But, because real terrain is not flat at all, the terrain cannot be simply represented with one angle. Therefore, in order to apply the above body control method, a piece of terrain is converted into one equivalent flat imaginary plane before it is used for body control. The conversion method is presented in the next section.

5.3.2 Estimation of Support Plane

A simple way to obtain a flat imaginary plane model for arbitrary terrain is to average all of the terrain data. However, averaging the terrain data for eliminating terrain irregularity leads to two problems so that this process is practically unsuitable for implementation with a moderate sized computer. The first problem is to determine how large an area should be used for this process, and the second problem is how many data points should be selected (or how to select data points) for the process. From the fact that the vehicle deals only with localized terrain, the area to be considered may be limited to a reasonable size, but a method to select a small subset of terrain data points is not simply determined. Therefore, in this study, the imaginary plane representing the terrain is derived from the footholds of supporting legs, not only because these points are meaningful for planning the motion of the vehicle, but also because the number of those points is not greater than six if a hexapod walking machine is considered. Because an imaginary plane is derived from the supporting points, in the following text, the term, support plane, is used instead of imaginary plane.

If the number of supporting legs is less than or equal to three, a support plane is easily found by drawing a plane which includes all the supporting points.
Practically, three supporting legs are the minimum to maintain static stability for
the ASV.) Though finding a support plane is easy in the above case, if the number
of supporting legs are greater than three, it is not always possible to make a flat
two dimensional support plane by including all supporting points. In this case, a
support plane may be obtained by linear regression on the supporting points [33],
which is utilized in this study. This procedure is referred as estimating a support
plane here.

If directly available data, that is, only supporting points, are utilized for
estimating a support plane, then the number of data points for the ASV can
be varied from three to six because three legs are the minimum number of legs
in order to be statically stable. This method is straightforward, but a different
approach is introduced here for the following reasons. First, a larger number of
data points are desirable to better estimate the actual terrain slope. Second, the
support plane derived from only supporting points will be frequently varying when
terrain becomes complicated because supporting legs may be frequently changed
in order to adapt to irregularity of the terrain. Third, the estimated support plane
is used to plan body motions at one sampling time into the future as well as an
additional prediction of one second. The predicted motions and terrain preview
information are utilized by the free gait algorithm to determine leg sequencing.
Thus, both present supporting points and desired predicted supporting points
are equally important pieces of information to the vehicle for the support plane
estimation process.

From the above considerations, by selecting one point from each leg, six points
are used as data points for estimating a support plane in this study regardless of
whether a leg is on the ground or in the air. Because the foot of a leg has an actual
support point only when it is on the ground, a point must be selected for a leg in
the air. If a desired future supporting point is used for the data point of its leg, then all six legs can be associated with six corresponding points.

Specifically, when a leg is in the *Ready* state, its desired future supporting point is determined by the free gait algorithm. (A detailed representation of the method to find the desired future supporting points will be introduced in Chapter 6.) This point information is continuously used for planning the motion of the leg in the *Advance* state and in the *Descent* state. After the leg contacts the ground, the current supporting point of the leg becomes available. Before lifting the leg from the ground, this current supporting point information is continuously available. But when the leg is lifted and returned to the *Ready* state, support point information is not available because of two reasons. The first reason is that the foot of the leg is no longer on the ground, and the second reason is that the free gait algorithm is not ready to determine its desired future supporting point.

In this study, during the *Lift* state and the *Return* state, the position occupied by the leg at the last moment of the *Support* state is utilized as the data point for the estimation process because this point is easily obtained inside the program. Moreover, this choice can eliminate frequent changes of the data points because the usage of the same position data used in the *Contact* and the *Support* states is extended through the two foot velocity control states, *Lift* and *Return*.

If there is no foot slippage during the *Contact* state and the *Support* state, and the supporting point of the leg is observed from the earth coordinate system, then the current supporting point of the leg is not changed. This is true in the *Lift* and the *Return* states because, from the previous discussion, the same supporting point is used for simplifying the process. Thus, the supporting point data of the leg is unchanged from the *Contact* state to the *Return* state as long as this point is expressed in earth coordinates. Because the desired projected supporting point
of the leg is determined in the Ready state, and the leg moves to the desired point during the Advance and the Descent state, the desired position data is unchanged if the position is expressed with respect to earth coordinates. After the leg touches down on the ground, the desired position becomes its current supporting point. Therefore, the two types of points of the leg have the same numerical value in earth coordinates. In other words, the numerical values of the data which are used for estimating a support plane are updated only in the Ready state during one complete leg cycle when a new desired future supporting point is assigned to the leg by the free gait algorithm. Consequently, the data utilized for the estimation process needs to be updated only when the free gait assigns a new desired future supporting point to a leg in the Ready state.

5.4 Summary

In order to increase the mobility of the vehicle, the motion of the vehicle body should be planned based on the characteristics of the terrain as well as the command from a human operator. For eliminating abrupt changes of the human command input, a low pass filter is used with 0.5 second time constant. Body attitude and altitude are controlled using a support plane derived from terrain data. Specifically, six data points selected from the six legs are used to determine a support plane to control the body motion. Though the number of data points may be more than six by including the previous support points [57], the previous data points are less important compared with present supporting positions or predicted supporting position, because locomotion of a legged vehicle by the free gait algorithm is made by projection mainly with terrain preview information.

Up to this point in this dissertation, only the separate control of body and legs has been considered. While this approach clarifies some of the issues involved
in terrain adaptive locomotion by a legged vehicle, body and leg motion cannot in fact be physically separated. Rather, \textit{coordination} of the eighteen controllable degrees of freedom provided by the leg actuators is required in order to obtain a desired six degree of freedom behavior in the ASV body. The next chapter of this dissertation addresses this problem.
CHAPTER 6

Free Gait Motion Coordination Algorithm

6.1 Introduction

In order to implement a free gait algorithm, three more topics remain to be discussed besides the trajectory and the states of a leg and the body control scheme already discussed in Chapters 4 and 5 respectively. In this chapter, the first topic is how to find a foothold for each leg which is in the air and is able to be used by the free gait algorithm for supporting the vehicle body. The criteria for foothold selection are the leg temporal kinematic margin, which will be formally defined in the next section, and the vehicle stability margin. The next topic is to describe the Leg_motion_planning_state program block. This block is located between the foot velocity control state block and the free gait algorithm in the sense of the logical hierarchy of the free gait program. Thus, this block controls the foot velocity control states by sending commands, and is controlled by the free gait algorithm. Therefore, the overall logical hierarchy of the free gait program is as illustrated in Fig. 6.1. Because the planner always makes predictions one second into the future, there exists a time difference between the planners (the upper and the lower level planners) and the execution blocks. In this hierarchical structure, the internal state of a block can be freely monitored by its parent block or by the blocks at the same level as that of its parent block, while only its parent can send state transition messages to it. Moreover, it continuously monitors the internal
Figure 6.1: Overall logical hierarchy of the free gait simulation program. (There exists a one second time difference between the planners and the execution blocks.)
state of its child block so that it can make proper state transitions. Consequently, the internal state of a block is updated by the block itself based on the message from its parent block or the internal states of its child block and the blocks in the same level as that of its child block. The third topic is the deceleration plan of the vehicle. This feature of the free gait algorithm gives the vehicle an extra freedom to avoid obstacles by allowing changes of speed without modifying the path determined by an operator.

Finally, the free gait algorithm, which utilizes all the ideas of Chapter 4, 5, and part of Chapter 6, is described in detail with several flow charts which are organized hierarchically. The whole program is currently running on a Symbolics 3640 LISP Machine. The free gait stepping algorithm portion of this program is written entirely in Common LISP [58].

6.2 Foothold Selection
6.2.1 Temporal Kinematic Margin

Though a spatial kinematic margin is proposed by McGhee [10], such a kinematic margin is not suitable in this study because of the difficulty of applying the concept to a vehicle which can turn in place. Instead of measuring a distance at the center of gravity in order to find the spatial margin, a distance which is measured from the current supporting point to the boundary of its working volume in the direction of the movement of the leg can overcome the problem which occurs when the vehicle makes a turn in a place action because each foothold can move a certain length of a segment of its trajectory. But the distance associated with each foothold cannot be compared in the same units. The distance of a foothold which is close to the center of gravity is slowly reduced if compared with that of a foothold which is far from the center. Therefore, the distances should be normalized so
that they are compared in the same units. To normalize the distance associated with each foothold, the distance is divided by the effective velocity measured in body coordinates. Thus, the unit of the result becomes a time, and is called the *temporal kinematic margin*. The formal definition is as follows:

**Definition 5**: A *temporal kinematic margin* is associated with each foothold. This margin is the time remaining at any given time until the associated leg would reach the boundary of its working volume if the foothold were used as a support point [11,34].

### 6.2.2 Conflicting Foothold Selection Criteria

A "best" foothold for a leg is one which in some sense maximizes both the temporal kinematic margin and the stability margin when the foothold is used as supporting point because making both margins larger gives better maneuverability to a legged vehicle. Unfortunately, an attempt to maximize both margins may lead to a contradiction because an effort to make the temporal kinematic margin larger often makes the stability margin smaller [11].

One possible way to deal with this contradiction is by devising an evaluation function which has two arguments, temporal kinematic margin and stability margin, so that footholds can be compared with one-dimensional values, but conceptually there is no clear way to define a good evaluation function. Moreover, even if a good evaluation function were to be developed, evaluating and comparing all possible cases demands considerable computation time because the complexity of the problem usually well exceeds several hundred possible cases, and is easily expanded to several thousand cases. For example, on an open terrain having a small percentage of forbidden cells the number of cases easily approaches the upper
bound 4913 or $17^3$ possible cases. This number results when three legs are in the 
*Ready* state and all the legs are assumed to have sixteen available footholds, which 
is the maximum number of available footholds for each leg using the algorithm to 
be developed later in this chapter. Since each foot could also be maintained in 
the ready position (not placed), there are a total of 17 possibilities for the future 
location of each foot. Since each foot can be moved independently of any other, 
$17^3$ cases results. Even if as much as 50% of the possible footholds are forbidden 
cells on the average, there will still be $17^3/8$ or 614 combinations of footholds to be 
evaluated. This complexity may not be well suited to a moderate-sized computer 
which can be carried by a vehicle such as the ASV.

One noticeable fact is that finding a foothold having the largest temporal 
kinematic margin is a locally confined problem for each leg because the margin 
can be calculated and compared without consideration of other legs, while finding 
a foothold having a largest stability margin leads to a globally interrelated problem 
among legs, because all the currently supporting leg positions contribute to the 
stability margin associated with a given foothold. Using the above mentioned 
characteristics, in this study, the foothold selection problem is decomposed into 
two parts: the one leg level and the other inter-leg level. The first part involves 
solving the localized problem; that is, to find the foothold which has the largest 
temporal kinematic margin among the available footholds of a given leg. The 
foothold found in this part is assigned to the leg. Thus, in the inter-leg level it can 
be considered that a given leg has only one foothold which is assigned in the first 
part.

In the second part, a leg is selected which can give the largest stability margin 
among the legs when the leg is stepped on the foothold assigned in the first step. 
If the desired stability margin cannot be achieved, then the algorithm returns to
the first step and resolves the localized problem again while the previously selected foothold is excluded, and this new solution is attempted again in the second step. If the desired stability margin is not obtained in the second step, the first problem is attempted again. Thus, these two-steps are repeated until all the remaining available footholds are tested or the desired stability margin is achieved with a foothold assigned in the first step. While details of the second problem will be discussed in a later section, only the methods to solve the first problem will be discussed in the next section and will be called “assigning a foothold to each leg”. The method to solve the second problem is more complex and occupies most of the free gait algorithm.

6.2.3 Trial Foothold Assignment for Each Leg

In order to develop a method for the tentative selection of footholds for each leg, some additional definitions are useful.

Definition 6: A reachable area is associated with each vehicle leg. This area is defined as the intersection between the working volume of the vehicle leg and the ground which supports the vehicle. The reachable area may be varied by changing the altitude of the center of the body or the attitude of the body. For each leg, on rough terrain, there may exist more than one disjoint reachable area resulting from the effects of rocks, holes, etc.

Definition 7: A constrained reachable area is associated with each vehicle leg. This area is defined as the intersection between the constrained working volume of the vehicle leg and the ground which supports a vehicle. The constrained reachable area may be varied by changing the altitude of the center of the body or the attitude.
of the body. Again more than one disjoint reachable area may be associated with one leg.

Definition 8: A \textit{maximal size constrained reachable area} is associated with each vehicle leg. This area is defined as the intersection between the constrained working volume of the vehicle leg and the flat level terrain when the vehicle body is parallel to the imaginary terrain and all the legs are extended to the limit of the constrained working volume. Only one unique maximal size reachable area exists for each leg.

Definition 9: A \textit{estimated reachable area} is associated with each vehicle leg. This area is defined as the intersection between the constrained working volume of the vehicle leg and the estimated support plane. The estimated reachable area may be varied by changing the altitude of the center of the body or the attitude of the body. Unlike a reachable area of a leg, this area is always one contiguous area for each leg.

If terrain information is kept with respect to an earth coordinate system, there is no need to update the terrain information during vehicle movement, while continuous updating is necessary when terrain information is kept in body coordinates. This is also true for a foot position on the ground because its position is stationary in earth coordinates as long as the foot is not lifted or does not slip on the ground during a movement of the vehicle body. Therefore, in this study, for simplifying the simulation, all terrain information and foot positions are internally stored in the earth coordinate system, and an ideal terrain is assumed on which the feet do not slip.

For speeding up the algorithm, another simplification is made which limits the number of search points for each leg to sixteen points out of the infinity of possible
points contained in the maximal size constrained reachable area of one leg. Fig. 6.2 shows one possible way to put the maximum number of terrain search grids inside the reachable area, with a grid dimension of one foot by one foot. In order to deal with uncertainties of leg movement, the sixteen grids are positioned in the middle of the area, and sixteen points are selected by choosing a middle point of each grid as reference points for searching a foothold. These sixteen points are named as the initial sixteen possible footholds (ISPF's). For each leg, the ISPF's are searched to find a foothold which may be used as an actual supporting point if the stepping algorithm decides to use it.

On generalized terrain, a reachable area may not have the same dimension as the full size reachable area shown in Fig. 6.2, which may be smaller and distorted. For example, if a valley exists under a leg, and the depth of the valley is deeper than the maximum height of the constrained working volume, the reachable area may be divided into two disjoint reachable areas for one leg. When there is a terrain with several sharp and deep edges under a leg, the situation can be worse so that more than two disjoint reachable areas exist for only one leg. For overcoming these difficulties, the following steps are devised.

First start from the initial sixteen possible footholds (ISPF) for each leg. When the free gait program is started, these ISPF's are stored inside the program because their values do not change in the body coordinate system. When a foothold selection process is necessary, the program immediately starts a foothold search with the pre-stored ISPF's if only the name of the leg is known to the program. Second, find an estimated reachable area using an estimated support plane as discussed in Chapter 5. The size of this area is always less than the full size constrained reachable area because the bottom of the constrained working volume has the largest area among any possible cut of the working volume using a flat surface. Third,
Figure 6.2: Full size reachable areas of six legs and initial sixteen possible footholds

Units: ft
the estimated reachable area is projected on the maximal size constrained reachable area, and estimated possible footholds of the leg are found by choosing from the ISPF's only points inside the projected estimated reachable area. Usually the number of the estimated possible footholds (EPF's) are less than sixteen which is the upper limit of the EPF's. (The number sixteen is the number of the ISPF's.)

Fourth, the z axis value of the estimated possible footholds is replaced with the z axis values obtained from the intersection point between the estimated support plane and lines which are parallel to the z axis of the body coordinate and passing through the EPF's. Now the points are on the estimated support plane and obtained in the body coordinate system.

The next step is to find possible available footholds (PAF's) on the terrain. First, each EPF is converted into the earth coordinate system, and then projected onto the imaginary horizontal plane in the direction of the z axis of earth coordinates. Second, the permitted terrain cells determined by the terrain scanner are projected on the horizontal plane in the direction of the z axis of earth coordinates. The horizontal plane thus has projected EPF's and projected terrain cells. The projected terrain cells have dimensions one foot by one foot because of the definition of the terrain model, but because the estimated support plane is not usually parallel to the horizontal plane, the effective size of the grid drawn on the estimated support plane is smaller by a factor of the cosine of the angle between the two planes, though the original sizes of both grids have the exact same size. Thus, there is a possibility that one projected terrain cell can contain more than one projected EPF's. Therefore, the third process is to find those projected permitted terrain cells which contain at least one projected EPF. The reason to find permitted cells instead of projected points on the terrain is that it is a design decision that the vehicle steps on the center of the terrain cells when it walks on
them. Fourth, using the permitted cells found in the third step, the possible available footholds (PAF) are selected from the center points of the permitted terrain cells on the terrain surface. These points are candidates for footholds for the leg because they are under the estimated reachable area, but there is no guarantee that all the PAF’s are inside the constrained working volume.

The next step is to check whether the PAF’s are in the constrained working volume. First, the possible available footholds are changed to the body coordinate system, because they are expressed in earth coordinates, and because the constrained working volume is defined in body coordinates. If footholds are picked from PAF’s which are inside the constrained working volume, available footholds are obtained.

The final step is to select a foothold which has the largest temporal kinematic margin among the available footholds, and then to assign the position of the foothold to the leg after converting the coordinate system from body coordinates to earth coordinates. Before finishing the last step, the available footholds excluding the footholds with largest temporal kinematic margin, are stored for each leg. Specifically, in this free gait program this information is stored in a leg structure. (A Common LISP structure is a collection of user definable fields and corresponding values which allow data abstraction [39,58]. It is similar to a “record” in Pascal.) If an adequate stability margin is not associated with the foothold assigned to the leg, only the final step of foothold selection needs to be repeated using the available footholds stored in the leg structure without repeating the long procedure to find available footholds for each leg. Because the set of available footholds stored in the leg structure does not include the previously assigned foothold, and because a foothold is chosen which has the largest temporal kinematic margin among the available footholds stored in the leg structure, the newly selected foothold which
has the largest temporal kinematic margin will be different from the previously assigned foothold. Therefore, if only the last step is repeated, in each repetition, the leg can be assigned successive footholds among its available footholds in the order of magnitude of their temporal kinematic margins.

Because the free gait algorithm runs with a one second predicted future time frame, before starting these procedures, the body position should be projected so that the initial possible footholds which are expressed in body coordinates can be correctly translated into earth coordinates. Specifically, the projected body position used in this implementation is the 0.6 second future projected position with the input velocity commands assumed to be the present velocity input commands during this period, and when both body control schemes are activated as described in Chapter 5. Because of the foot trajectory chosen in Chapter 4, the values of the x and y axes of the foot position of a leg in the earth coordinate system are not changed when the foot velocity control state is in Descent, and this period is planned as 0.4 seconds. Therefore, there is a possibility that a leg of the vehicle cannot be positioned at a point which is 1.4 feet above a foothold which can be included in the constrained working volume of a leg when the vehicle reaches the one second projected body position, but which cannot be included in the constrained working volume of a leg when the vehicle reaches 0.6 second projected body position. That is, even though a given foothold may be reachable 1 second in the future, ground tracking of that point may not be possible 0.6 seconds in the future. Since the selected foot trajectory requires ground tracking, this phenomenon can have the effect of reducing the number of terrain cells considered for footholds for a given leg.

Exactly the opposite situation to that described above may exist when the terrain is sufficiently complicated. When a region of the terrain is easy for the
algorithm, the algorithm will use footholds which are located in the direction of
the vehicle movement. That is, when the vehicle is moving forward, the algorithm
will choose a foothold which is located at the front side of its reachable area. If
a foothold is chosen which is near the far back side of the reachable area formed
by 0.6 seconds projected body position because the point is the only permitted
foothold in the area, the foothold may not be reachable when the leg touches
down on the foothold if the foothold is not included in the reachable area formed
by the one second projected body position.

Because of the two opposite situations described above, in the implementation
of this dissertation, footholds are searched by utilizing the 0.6 second projected
body position, and if the temporal kinematic margin of a foothold is less than
the planned contact time, 0.4 seconds, it is removed from the available.footholds
set. On the other hand, to obtain the correct temporal kinematic margin at foot
touchdown, one second projected body velocities are used in order to calculate the
margin instead of 0.6 second projected body velocities because the velocities of
foot on the ground are determined when a leg touches down on the ground; i.e.,
one second after planning is performed.

6.3 Leg Motion Planning State Diagram

Differing from a wave gait, a free gait algorithm does not have a predeter-
mined leg sequence. The leg sequence should be determined by a free gait algo-
rithm considering environmental factors, such as presently supporting legs, terrain
information, and desired body movement provided by a human operator or by
an autonomous navigation system. In order to plan leg sequencing smoothly and
efficiently, in this study the planner is made to plan with a one second prediction
time based on the reasoning explained below.
Though planning can be done inside the vehicle computer in a relatively short
time, the physical legs cannot immediately follow the planned motion because
of mechanical limitations. For example, even though the planner decides to use
one more leg to support the body at a certain time, the actual physical leg cannot
support the body immediately on the arrival of the planner's decision. This latency
problem may be solved by early planning so that the planner sends a decision
earlier than the time that the actual touch-down event is desired. Basically, the
lower bound of the prediction time is the stepping time needed by a leg when
a decision from the planner is given to it until it touches down on the terrain.
Though leg sequencing planned a longer time ahead than the time required for a
leg to step seems desirable, such an approach requires more computation power
and is also subject to greater inaccuracy due to errors in predicting commands
from the operator. Therefore, in this study, the prediction time is set at the lower
bound, one second.

In order to allow a planner to plan one second into the future, two pieces of
information are required. First, a one second prediction of body position including
altitude and attitude, is necessary because this information provides a basis for
calculation of the stability margin and temporal kinematic margins for supporting
legs. In this implementation, the future position is estimated based on the current
desired operator commands assuming that the desired input will not change within
the next one second. For making this assumption reasonable, as explained in
Chapter 5, operator commands are filtered with a low pass filter before applying
them to the free gait algorithm. This approach reduces the position difference
between the one second estimated body position and the actual position where
a vehicle will be one second later. The second piece of information is terrain
preview information. This information is utilized in order that the vehicle can
classify terrain cells as suitable or unsuitable. In the case of ASV, this type of
information is gathered by the optical radar mounted on the top of the vehicle
cabin [14]. The gathered data are not predicted in the same sense as the future
body position. That is, if a desired future stepping position is selected and stored
in the earth coordinate system, then the vehicle can correctly step on the desired
position even with small deviations between the planned body position, and actual
body position. Therefore, this implementation can generate fairly reasonable leg
motion planning with predicted body positions and terrain preview data ahead of
the vehicle.

For leg motion planning, a finite-state machine is devised whose state diagram
is shown in Fig. 6.3. This machine is on the first level above the foot velocity
control state block. Because this machine is a Moore machine, labels on arcs from
state to state are transition conditions, while arrows not terminating on states
represent outputs. This is not the usual way to draw outputs for Moore machines,
but is included here to emphasize communication to the lower level leg velocity
control machine described in Chapter 4. For simplicity, outputs to the upper level
planner are not drawn. This convention reflects an assumption that upper levels
of software have access to status information at lower levels, but those lower levels
must receive explicit commands from upper levels to update their internal status.

Differing from the foot velocity control state diagram, this state diagram has
only one type of state, the asynchronous type. This block controls the velocity
control states by sending two commands, the \texttt{Deploy\_command} and the \texttt{Re-
cover\_command}, and also monitors the activities of the velocity control states in
order to use them as terminating events of its states. Specifically, the \texttt{Ready\_state},
\texttt{Contact\_state}, \texttt{Support\_state}, and \texttt{Interlock\_confirm} signals are monitored from the
foot velocity control blocks. The former three conditions are actually the names
Figure 6.3: Leg motion planning state diagram for free gait algorithm. (Planner runs 1 sec. in future. Assumes current joystick input remains constant for 1 sec. in to future.)
of the states monitored from the foot velocity control block associated with the motion planning block, but the last event, Interlock-confirm, is monitored from two distinct foot velocity control blocks because Interlock-confirm is an event involving two different legs, and each leg has its own distinct foot velocity control block and leg motion planning block. How to test the Interlock-confirm condition will be discussed later in detail.

The block shown in Fig. 6.3 also interacts with the higher level software, the free gait planner. The planner sends a terminating condition to this block, which is one of Place-decision, Lift-decision, and Exchange-decision. The Place-decision command is sent when the leg associated with the leg motion planning state block is necessary to support the body, while the Lift-decision or the Exchange-decision commands are sent to this block when the leg is no longer necessary to support the body. The details of this action will be presented in later paragraphs.

When the foot velocity control state is changed to Ready, the leg motion planning state is changed to the Available leg state. This state means that a given leg is available to the free gait planner. Because one distinct leg motion planning state block is associated with each leg respectively, the planner should check all the six leg motion planning state blocks to find which legs are available in order to support its body one second in the future. While the leg motion planning state is Available leg, the leg is waiting for the Place-decision command from the upper level planner. If the planner decides to use the leg to support the body, the state is changed to the Planned contact state by this decision. At the same time, the leg motion planning state block sends the Deploy command to its foot velocity control block. For the planner, it is assumed that the leg is already supporting the body because the planner predicts one second in the future.

The leg motion planning block waits until the foot velocity control state is
changed to the *Contact* state. As soon as the state is changed to the *Contact* state, the leg motion planning state is changed to the *Eligible to lift* state. In other words, when the leg physically touches down on the ground, this leg motion planning state transition occurs. If the *Descent* state lasts for 0.4 seconds, which is the planned time amount for this state as discussed in Chapter 4, then the *Planned contact* state of the leg motion planning state lasts one second. Thus, one second projection time is spent while the leg motion planning state block waits in the *Planned contact* state, and the leg motion planning state diagram is synchronized with the physical touch-down event. The *Eligible to lift* state notifies the planner that the leg associated with the leg motion planning block is ready to be lifted from the ground. Therefore, the planner will add this leg to its leg name list which can be removed from a support pattern whenever the planner decides to do that. Removal of a leg from a support pattern by the planner is performed in two different ways. One way is just deleting a given leg. In the second way, when the planner removes a given leg, it will add another leg into the support pattern. The latter method makes the control scheme complicated, but improves the mobility of the vehicle. It is observed that most of the time the free gait algorithm uses the latter type of leg sequencing when the vehicle is walking on a flat level terrain without forbidden cells. A detailed discussion will be given in the following paragraphs.

First, for discussing the latter case, it is assumed that three legs are in a support pattern, but one leg is almost at its kinematic limit. If the leg near its kinematic limit is simply deleted from the support pattern, the vehicle will become unstable because a minimum of three legs are necessary to maintain the static stability of the vehicle. On the other hand, unless the leg is lifted from the terrain, the vehicle will soon come to stop because of the leg near its kinematic limit. There-
fore, this situation cannot be solved by removing only one leg without stopping the body. One way to solve this problem is to use two legs instead of one leg. That is, one leg selected from the list containing names of legs in the Available leg state is made to support the body, and at the same time another leg near its kinematic limit is commanded to be lifted. These two actions are performed simultaneously inside the planner, but the planner sends two distinct decisions to corresponding leg motion planning state blocks. First, the Exchange decision is sent to the leg motion planning block of the leg to be lifted so that the state is changed to Planned exchange. Second, the Place decision is sent to another leg motion planning state block so that the leg starts to move in order to eventually support the body. The former leg motion planning block will wait until the Interlock-confirm signal is received. This signal is generated when the leg moving toward the supporting point touches down on the ground. After arrival of the confirm signal, the state of the former leg motion planning block is changed to the Actual lift state. Because the output of the Actual lift state is the Recover command, on reception of the Recover command the state of the foot velocity control block will be changed to the Lift state; i.e., the leg associated to the command is physically lifted from the ground.

Besides the Interlock-confirm signal, the Support state signal from the leg to be lifted is also tested before making the above leg motion planning state transition. This is done to ensure that the delay from entering the Planned exchange state to exiting this state is never less than 1 second. That is, in case the contact signal from the leg to be placed arrives in less than one second after that leg receives the Deploy command, the lifting of the other leg could occur prematurely. This in turn could have an adverse effect on stability because it represents a deviation from the planned motion.
Referring to Fig. 6.3, the Actual lift state of the leg motion planner can be entered either from the Planned exchange state or from the Planned lift state. The first case is discussed above. With respect to the second case, consider a situation in which four legs are in a support pattern (or in Eligible to lift states), but one leg is almost at its kinematic limit. Suppose that without the leg which is near its kinematic limit, the vehicle can be statically stable. In this case, simple deletion of the leg is possible without harming the stability of the vehicle. Therefore, the planner sends the Lift.decision to the leg motion planning block so that the leg motion planning state is changed to Planned lift, and the leg is removed inside the planner so that the leg no longer exists in the predicted support pattern of the planner. But the physical leg cannot be lifted from the terrain as long as the foot velocity control state is the Contact state. The leg motion planning block therefore waits until its state is changed to Support state. After this change is detected, the state of the leg motion planning block will be changed to Actual lift since it has been assumed that the additional stability test is satisfied.

Next consider the case in which the planned leg state (1 second in future) involves four supporting legs, but the physical (current) state involves only three such legs. This can happen when one of the four legs in the planned supporting pattern is not in the Eligible to lift state but in the Planned contact state. In this case, the leg with the Planned contact state is moving toward its desired supporting point on the ground, but it is still in the air. To the planner, which always predicts future events, four legs are in its supporting pattern. Thus, without the leg near its kinematic limit the vehicle can be stable. Therefore, the planner generates the Lift.decision, and sends it to the leg motion planning block. If the leg near its kinematic limit were in the Support state and Support.state were the only terminating event to make the state transition, then the state would be changed
to Actual lift as soon as the Lift\_decision arrived, and the new state, Actual lift, would generate the Recover\_command so that the physical leg associated with the leg motion planning block also would be immediately lifted. This lifting action could cause momentary instability of the vehicle until the leg in the Planned contact state actually touched down on the ground. Therefore, testing only one condition, Support\_state, before making a transition from Planned lift to Actual lift, is not a sufficient criterion. As shown in the state diagram, both Support\_state and the above touch-down event should be checked for ensuring a correct state transition. In this program, the touch-down event is checked by testing the current stability because the name of the leg in the Planned contact state is not directly known to the leg in the Planned lift state, and because the touch down event can be checked by the current stability test.

Differing from leg exchange in which two legs are synchronized during lifting and placing, in the above described planned lift case, two legs are coupled by the condition that one leg cannot be lifted before placing the other leg. An action to place one leg is already started before the state of the other leg is changed to the Planned lift state. On the other hand, in leg exchange, the leg placing action is started when the state of the other leg is changed to Planned exchange. Therefore, the leg exchange action can be considered as the limiting action of the case of lifting one leg. For efficient operation of the free gait algorithm, the leg exchange action is included and it is observed that this action frequently occurs when the vehicle uses a tripod-like gait pattern on a flat terrain containing small number of obstacles.
6.4 Body Deceleration Methods

Just smoothing the velocity commands as described in Chapter 5 is not enough to control the vehicle in extreme terrain conditions. That is, if the density of permitted terrain cells is sufficiently low, there is a possibility that the vehicle cannot follow the output velocities from the velocity smoothing schemes discussed in Chapter 5. Specifically, the limitation on the vehicle mobility is due to two types of problems, timing or permanent. The latter type is known as a deadlock condition.

Whenever the mobility of the vehicle is limited, the free gait algorithm treats this unfavorable condition as a timing problem even though it may turn out to be a permanent problem. If the unfavorable condition detected is a timing problem, then it will be automatically solved by the free gait algorithm. If the condition is not in the first category, then the algorithm will fail after applying all possible methods that the algorithm can use to overcome it during a certain period of time. Thus, the algorithm automatically classifies the condition as the latter category, and makes the vehicle stop in a deadlock condition. When this happens, the only way to continue the vehicle motion is to choose a new path over the terrain.

Specifically, unfavorable conditions can happen either when the stability margin is less than the safety margin or when the temporal kinematic margin of a leg is less than the specified safe kinematic margin. As explained in Chapter 3, both margins can be set by a human operator when this program is started. Sometimes both problems can contribute to cause an unfavorable condition. Generally, the free gait algorithm can avoid unfavorable conditions by adding one more leg in the support pattern for increasing the stability margin or by lifting a leg near a kinematic limit to eliminate its kinematic problem. Because the planner predicts one
second future events, these actions can be done without interrupting the mobility of the vehicle. But if the addition or deletion of a leg is not possible, the vehicle confronts an unfavorable condition. The first approach to dealing with this condition is slowing down the speed of the vehicle so that the predicted unfavorable situation may not actually happen one second after the prediction is made. In other words, though the distance to the problem spot is not changed, the traveling time to that location is prolonged because the algorithm reduces the present commanded velocities. Therefore, if the problem is temporal, the algorithm can solve it during the newly prolonged amount of time before the vehicle reaches the spot. If the algorithm cannot find a suitable leg to solve this problem, it gives up the solution process and declares the problem as a permanent problem by entering a deadlock state.

In order to slow down the vehicle when unfavorable terrain conditions are encountered, the operator commanded velocity is divided by an integer, n, called the deceleration factor. This scheme is illustrated on Fig. 6.4. Initially, n is set to one. When attenuation of the operator commanded velocity is desired, n is increased to k. If this does not permit footholds to be found, n is again increased by k, where k is a factor selected by the operator. This process continues until either a satisfactory foothold is found or until the vehicle is deadlocked. To ensure that deadlock ($V_0 = 0$) actually occurs, a deadzone nonlinearity is connected in series with the output of the integrator of Fig. 6.4.

When deadlock does not occur, the process described above is reversed after a foothold is found so that the deceleration factor is decreased by an amount of k at each cycle until it eventually is restored to one. A remaining problem in this scheme is to select a value for the deceleration factor increment, k. In this research, k was arbitrarily chosen to have a value of 2. While this was found to
\( n = \text{deceleration factor} \)
\( n \in \{1, k, 2k, \ldots, 100k\} \)

Deadzone values:
\( \pm 0.02 \) for \( v_x, v_y \)
\( \pm 0.005 \) for \( r_z \)

Figure 6.4: Complete velocity control regulation scheme
produce generally acceptable results, the possibility of using other values for k, or even an entirely different approach to achieving vehicle deceleration, remains to be investigated. Since this question is not central to this dissertation, it has not been further explored by the author.

6.5 Flowchart Representation of Free Gait Algorithm

A flow chart for the overall free gait simulation program is shown in Fig. 6.5. At the beginning of the program, a sample terrain is created for the program by a human operator or a terrain generation program. This process involves two sub-tasks. The first task is to make an elevation profile for the terrain. This is done by the human operator. The second task is to provide obstacles on the terrain. This can be done either manually or automatically by using a random number generator. In the latter case, the percentage of forbidden regions of the whole terrain should be assigned by a human operator [11]. After finishing the creation of the sample terrain, the program starts to initialize simulation data, such as body attitude and altitude, and leg positions, in the “Initialize simulation” block. When the initialization process is finished, the program enters the main loop which will be repeated until the human operator wants to stop the simulation program.

In the main loop, the first step is to read the desired velocity commands from outside the program. Based on these commands, the program plans the motion of the vehicle in the “Plan motion” box where the free gait algorithm is implemented. After the planning process is finished, the planned motion is executed in the “Execute motion” box. Specifically, the “Plan motion” block sends to the “Execute motion” block the following information: Recover command, Deploy command, desired stepping position, and desired body position. Using the data generated from the “Execute motion” box, the “Display motion” box draws
Figure 6.5: Flowchart for free gait simulation program
the vehicle and the terrain on a graphic display terminal. If the human operator does not terminate this program, it then starts another loop with a new simulation time increased by 0.1 second.

The “Plan motion” block is refined in Fig. 6.6. This figure shows that whenever the vehicle possess adequate stability, an attempt is made to lift or exchange legs. Lifting is tried first. If this fails, then an exchange is tried. If this is not possible, no leg commands are generated. Evidently, this strategy tends to minimize the number of supporting legs. This in turn tends to maximize the number of legs in the Ready state and therefore available for finding new footholds in difficult terrain. If a stability margin is not larger than the safety margin, the algorithm tries to use one more leg for obtaining larger stability than the safety stability margin. In this case, the number of legs are increased for regaining the stability of the vehicle. When no such leg can be found, vehicle deceleration or deadlock results.

At the beginning of the “Plan motion” block, the free gait algorithm updates leg motion planning states for all six legs based on their own foot velocity control states. After the updating process, three lists are constructed based on the updated leg motion planning states of the six legs. The first list is constructed by including all legs which are in the Available leg state so that the higher level planner in Fig. 6.1 can select one of them whenever it needs an additional leg to support the body. This list is called the Available leg list in this dissertation. The second list is constructed by including all legs which are in the Eligible to lift state. The higher level planner can delete any leg in this list whenever leg removal from a predicted support pattern is necessary. This list is named the Eligible to lift leg list. The final list contains those legs which are in either the Planned contact state or the Eligible to lift state. In other words, this list contains legs in the predicted support
Figure 6.6: Flowchart for plan-motion block
pattern. Therefore, this list is used to calculate a predicted stability margin. This list is named the *Supporting leg* list.

In the "Calculate predicted leg and body position" program block, new body attitude and altitude variables are calculated based on the given velocity commands as discussed in Chapter 5.

In the third block, the program finds, for each leg in the *Available leg* list updated in the first block, a desired future supporting foothold which has the largest temporal kinematic margin among the possible footholds in the reachable areas of the leg. In this block, temporal kinematic margins of the legs in the *Supporting leg* list are also updated using the updated body position and velocities.

In the fourth block, the program tests which leg, if any, in the *Eligible to lift* leg list is at its kinematic limit. If a leg at its kinematic limit is found, the program removes the leg from the support pattern and increases the deceleration factor in order to reduce the speed of the vehicle. In a normal situation, this block is not executed, but sometimes this process is necessary. For example, consider a situation in which three legs are supporting the body, and the other three legs are returning to the *ready* position. Specifically, three legs are in *Eligible to lift* states, and the other three legs are in *Actual lift* states. If one of supporting legs has almost reached its kinematic limit, and the "Emergency lift for Leg m at kinematic limit" block were not included, then after executing the stability test, the algorithm would choose the left side block, "Try to lift a leg or exchange legs", because the vehicle is still stable. In the left block, the leg at the kinematic limit will be detected, and two possibilities will be tried: to lift the leg or to exchange the leg with another leg in the *Available leg* list. The first possibility does not work because the leg cannot be simply lifted from the ground due to the static stability criterion. The other possibility, exchange legs, cannot be used because there is no
leg in the Available leg list. The above problem can be solved by the "Emergency lift for Leg m at kinematic limit" block.

The situation mentioned above will happen one second later if the body movement follows exactly the planned trajectory. If this trajectory is changed in a correct way, the above situation may be avoided. In this program, only speed is reduced by some scaling factor, because modifying the path of the vehicle body by the stepping algorithm is not included in this study. Therefore, reducing the commanded velocity by the deceleration factor is a way to avoid the problem mentioned in the previous paragraph. However, although the vehicle speed is reduced, the leg which will be at the kinematic limit is still a bottleneck to the mobility of the vehicle, unless it is lifted from the ground. In order to achieve physical leg removal as soon as possible, the leg is deleted from the projected support pattern temporarily by this block. This leg removal from the pattern will force the program to choose a leg addition routine, the "Try to place one more leg" block, because the stability test will fail due to the removed leg which was essential to make the vehicle stable.

In the actual implementation, if a leg which is at the kinematic limit is found, then the deceleration factor is increased to decelerate the body velocity and the leg is temporarily deleted from the support pattern so that the deleted leg can be restored if the leg addition by the "Try to place one more leg" block fails. For safe operation, when the kinematic limit is tested, it is not checked at the actual limit of the working volume, but at the boundary of a working volume reduced by the distance that the leg associated with the volume can travel in a certain amount of time if the leg is continuously moving with the same speed at the time that the test is performed. The time margin is set as 0.4 seconds in the program.

The next step is checking whether the stability margin of the vehicle is larger
than the safety margin which is set for safety purposes a human operator. If
the margin is larger than the safety margin, then "Try to lift a leg or exchange
legs" block is executed. Thus, a Lift_decision or Exchange_decision command can
be generated so that a state transition from Eligible to lift to Planned lift or to
Planned_exchange is made in the leg motion planning finite-state machine of Fig.
6.3. If this test fails, then the "Try to place one more leg" block is executed. Thus,
the Place_decision command can be sent to the leg motion planning finite-state
machine of Fig. 6.3 so that a state transition from the Available leg state to the
Planned_contact state is possible.

The final block is the "Generate leg commands" block. In this block, the
Deploy_command and Recover_command as well as desired future footholds are sent
to the "Execute motion" box of the overall free gait simulation program shown in
Fig. 6.5, where the six foot velocity control finite-state machines are implemented.
Of course, on either the right or left side of this diagram, the attempt to lift or
place legs may fail. In that case, no leg commands are generated. In fact, this
is the usual result of the execution of this procedure since the execution rate of
the free gait algorithm should be substantially higher that the rate of stepping in
order to obtain smooth and effective vehicle control.

The detailed flowchart of the lift-exchange block is shown in Fig. 6.7. This
block realizes the "Try to lift a leg or exchange legs" function of Fig. 6.6. At
the beginning of this block, the deceleration factor is decreased because there is
no need to decelerate the vehicle speed. If the deceleration factor is equal to one,
then the factor is not decreased any more. The next step is to select Leg i which
has the smallest temporal kinematic margin among legs in the Eligible to lift leg
list, and delete Leg i from the Supporting leg list in order to check whether Leg i
is a redundant leg to make the vehicle stable or not. In the next decision block,
Figure 6.7: Flowchart of lift-exchange block
this redundancy test is performed by checking the stability without Leg i. Again the same safety margin is used to test the stability of the vehicle. If the stability test is satisfied, then Leg i can be removed without harming the stability of the vehicle body. In the program, the leg motion planning state of Leg i is changed to Planned lift. Therefore, the number of supporting legs for the vehicle is minimized in conformity with the general strategy of the free gait algorithm. If the stability test fails, then the “Try exchange” block is executed. The refinement of this block is shown in Fig. 6.8.

In the try-exchange block which realizes the “Try exchange” function of Fig. 6.7, there are two possible exits. The first exit from the block is to exchange legs, and the other exit is doing nothing with respect to leg sequencing. At the beginning of this block, the program searches over legs in the Available leg list to find Leg j which will give the largest stability margin among the legs in the list in case Leg j were added to the set of supporting legs. Specifically, the stability margin of a leg in this case is defined as the stability margin which results when the leg is placed on the terrain without changing legs in the support pattern excluding Leg i which was already removed form the support pattern in the previous block. Because Leg j is selected based on the stability criteria, if Leg j is placed, then the stability of the vehicle tends to be maximized by this action. However, this maximization is somewhat limited because at most three points can be compared to find Leg j, and each point has been selected from the available footholds of a leg by utilizing the criterion of temporal kinematic margin.

If Leg j is not found, then there is no choice other than to use Leg i again to make the vehicle stable. But if Leg j is found, then the program uses another test to determine whether or not the temporal kinematic margin of Leg j found in this block is larger than that of Leg i. If the above test succeeds, then a leg exchange
Enter

Search over available legs to find leg $j$ which maximizes SM if placed.

yes

Exists leg $j$ & Stability $> \epsilon$ if leg $j$ placed?

no

Is TKM of leg $j$ larger than TKM of leg $i$?

no

Leg exchange

yes

Restore Leg $i$ into support pattern

Exit

Figure 6.8: Flowchart of try-exchange block
is performed by changing the state of Leg j to *Planned exchange*, and the state of Leg i to *Planned lift*. This is shown in Fig. 6.9.

If Leg j has less temporal kinematic margin than that of Leg i, then Leg i is restored into support pattern. The restoring process is necessary here because of the following two reasons: First, the vehicle was stable before Leg i was removed from the support pattern. The process of restoring Leg i makes the vehicle stable again. Second, Leg i has the smallest temporal kinematic margin among legs in the *Eligible to lift leg* list. That is, Leg i represents a bottleneck for the vehicle mobility. If Leg j were to be used in spite of having less temporal kinematic margin than Leg i, it would make the kinematic problem more serious, and the vehicle would be stopped sooner than when Leg i is used for supporting the body.

Until now the “Try to lift a leg or exchange legs” block of Fig. 6.6 has been described in detail. Next, the right side block of Fig. 6.6, “Try to place one more leg”, will be described in detail. Because the stability test has failed, an additional leg is necessary to make the vehicle stable. The first refinement of this block is shown in Fig. 6.10. At the beginning of this procedure, the program tries to select Leg k which can give the largest stability margin among the legs in the *Available leg* list. This attempt may or may not be successful because any one of the legs in the *Available leg* list may or may not generate a suitable stability margin when it is used to support the body. If Leg k is found, and the stability margin produced by Leg k is larger than the safety margin, then Leg k is used to support the body by changing the leg motion planning state of the leg to the *Planned contact* state, and the deceleration factor is increased because the unfavorable condition is solved by Leg k. If Leg k cannot be found, or the stability margin of Leg k is less than the safety margin, then the program executes the right side block, the “Slow down vehicle or use new foothold” block.
Figure 6.9: Flowchart of exchange block
Figure 6.10: Flowchart of place block
The detailed flowchart of slow-down-vehicle-or-use-new-footholds block is shown in Fig. 6.11. This block realizes the “Slow down vehicle or use new footholds” function of Fig. 6.10. At the beginning of this block, the program checks whether all the legs in the air are in the Available leg state. This test can tell the reason why Leg k suitable to support the vehicle is not found in the upper block described in the previous paragraph. If this test fails, then there is a possibility that one of the legs in the air but not in ready position may make the vehicle stable. That is, one of the legs in Actual lift state may become Leg k when its state is changed to the Available state. In order to permit the leg to arrive at the position of the Ready state, some amount of time is required. During this period, the movement of the body is slowed down to maintain the stability of the vehicle because the vehicle is still stable at present. In addition to deceleration, if any leg is deleted before by the “Emergency lift for Leg m at kinematic limit” block of Fig. 6.6, then the leg will be restored into the support pattern to restore the stability because the vehicle cannot be stable without the deleted leg. However, this action will not cause a kinematic problem, because the speed of the vehicle movement is reduced by the deceleration factor in the next cycle. Therefore, the newly projected body position will not reach the problem point.

If all the legs in the air are included in the Available leg list, then no more new legs with Available leg state can be found. Therefore, there is no possibility of finding Leg k which can make the vehicle stable. In other words, the vehicle cannot be stable with any one of the footholds initially assigned to the legs in the Available leg list. However, the footholds assigned to the legs were selected from their available footholds based on the criterion of temporal kinematic margins of footholds, without considering the stability of the vehicle as discussed in the second section of this chapter. If other footholds among the available footholds of the legs
Figure 6.11: Flowchart of slow-down-vehicle-or-use-new-footholds block
in the Available leg list are used, then there is a possibility that the vehicle can be stable. Basically, this idea is implemented in the “Try new footholds” block which is executed when Leg k cannot be found. The detail of this block are shown in Fig. 6.12.

At the beginning of the block, the program tries to assign new footholds to the legs in the Available leg list. Thus, a new desired supporting point is assigned to each leg in the Available leg list according to the order of the magnitude of the temporal kinematic margin of its available footholds. That is, at the first try, all the legs in the list will be assigned with their new desired footholds which have the largest temporal kinematic margin among their available footholds while already assigned possible supporting footholds are eliminated from the list of available footholds. If all the possible footholds are already tried without making the vehicle stable, then the program will try the “Potential deadlock recovery” block at the left side of the flow chart.

If new desired footholds can be assigned to at least one or more legs, the program will search over the legs assigned with new footholds to find Leg p which can give the largest stability margin if placed among the legs which have newly assigned footholds. If Leg p gives a larger stability than the preset safety margin, then the state of Leg p is changed to Planned contact in order to make it to support the body, and the program flow will exit this block and return to the caller. If Leg p is not found, then the program flow will exit this block and return to the caller. If Leg p is not found, then the program goes back to the first box and tries to find a new foothold for each leg in Available leg list. This process will be repeated until Leg p is found or no more new footholds are found because all the possible supporting points are already tried.

A refinement of the deadlock-recovery block is shown in Fig. 6.13 which is executed when the previous attempt fails. This block realizes the “Potential deadlock
Find new foothold for each available leg,

Possible?

yes

Search over available legs to find leg p which maximizes SM if placed.

Exists leg p & Stability > ε if leg k placed?

no

Potential deadlock recovery

Place leg p

Exit

Figure 6.12: Flowchart of try-new-foothold block
Figure 6.13: Flowchart of deadlock-recovery block
recovery” function of Fig. 6.12. The central idea in deadlock recovery is to place more than one leg since deadlock results from the failure of leg lifting, leg exchange, and placing of a single leg. In this block, the first operation is to find and to assign footholds with the largest temporal kinematic margin among available footholds for the legs in the Available leg list, because available foothold information stored in the legs are destroyed in the previous block. The second operation is to find legs which have more than one possible supporting point, and to make a useful leg list by including the legs found. This process is necessary because some legs in the Available leg list do not have any available footholds due to the obstacles under them. If this operation succeeds, there exist one or more legs to support the vehicle, but by placing only one leg among the useful legs the vehicle cannot be made stable because all possible attempts to make the vehicle stable with one leg failed before executing the “Potential deadlock recovery” block. Thus, from the point of view of the free gait algorithm, all the legs in the useful leg list are identically useless in order to make the vehicle stable. Therefore, any leg from the legs in the list can be arbitrarily selected, and it can be used in order to support the vehicle without altering the general strategy of the free gait. In addition, the deceleration factor is increased in order to reduce the speed of the vehicle. This action will preserve the current stability of the vehicle, and allow that the free gait algorithm to find a suitable leg to make the vehicle stable in the next sampling time.

If a suitable leg is found in the next sampling time, it can be said that two legs could possibly make the vehicle stable at that time. Because the basic free gait algorithm is not designed to handle two legs at a time, by restarting it after adding an arbitrary supporting leg, the algorithm can have a chance to consider the next possibility, i.e., to find an additional leg utilizing the basic algorithm.
Therefore, the leg arbitrarily chosen to support the body acts like a seed to find a next reasonable leg. Though this method cannot find an ideal solution which could be generated by an algorithm able to deal with placing two or more legs at a time, this approach to the “Potential deadlock recovery” problem just described can be generalized so as to approximate such an ideal algorithm. Specifically, if the first trial which makes one leg support the body is not successful, then another leg is arbitrarily selected to be placed on the ground so that a solution can be attempted by the unmodified free gait algorithm in the next sampling time. This action will be repeated until all the useful legs are used. Because in this implementation, the maximum number of legs in the Available list is three, this block can be executed at most three times. After the three trials, if the free gait algorithm cannot find a reasonable foothold which can make the vehicle stable, there exists no leg in the useful leg list. Therefore, Deadlock occurs.

6.6 Summary

The main strategy of the free gait algorithm of this dissertation is to maximize the number of legs in the air so that these legs can be utilized when the vehicle mobility is limited either because of a small stability margin of the vehicle body or because of a small temporal kinematic margin of one of the vehicle supporting legs. To some degree, this strategy is complicated by the foot trajectory chosen in Chapter 4. Specifically, legs which are not in the Ready foot velocity control state cannot be utilized to restore the stability margin or to help to eliminate a leg near its kinematic limit, though they are in the air, because such legs are temporarily not available to the algorithm. Therefore, in order to solve this timing problem, some amount of time is needed which is long enough for their states to be changed to the Ready state before the vehicle meets an unfavorable condition. This is
possible because the algorithm always projects one second future events. For this purpose, the vehicle speed is decreased when necessary by the body deceleration methods developed in this chapter. Once such a timing problem is resolved, then the speed is accelerated until it reaches the desired velocities commanded by a human operator.

Because the algorithm always projects one second into the future, the projected leg events should be tracked and controlled by some means. In this study, this task is performed by the six leg motion planning state diagrams. Thus, these state diagrams simplify the task of the free gait algorithm in controlling individual legs because they automatically translate future decisions from the algorithm to present leg commands which can be directly given to their foot velocity control finite-state machine. Therefore, this discrete control system acts as a bridge between the algorithm and the foot velocity control state diagram. The leg motion planning finite-state machine of a leg observes the state of its foot velocity controller, and sends the Deploy_command or the Recover_command to that controller if necessary, while it receives one of three different decisions from the coordination algorithm, which are the Place, the Lift, and the Exchange decision, if the algorithm wants to send one. Additionally, it sends two types of information to the algorithm. One is whether its leg is available to the algorithm or not. The other is whether its leg is eligible to lift from the ground or not.

The foothold selection process is a separate problem from the above logical state transitions, but it indirectly affects them through the leg selection method of the algorithm. Specifically, the free gait algorithm compares two important values determined by the assigned foothold to a leg, and selects a leg which will support the body. These values are the temporal kinematic margin and the stability margin. As discussed in Section 6.2, in order to select a foothold for a leg, the former
margin is maximized first, and the latter margin is maximized later. If the latter margin does not meet the criterion that it is larger than the safety margin, the foothold selection process repeats the first step again while discarding the previously selected foothold. This process is repeated until a suitable foothold is found or all the available footholds are tested. This problem solving paradigm can be regarded as a generate-and-test system [52]. The generator of this problem solving paradigm is complete, nonredundant, and informed, which are the properties of a good generator [52] because all the available footholds are generated one after another in the order of their temporal kinematic margins. Moreover, all the available footholds are already limited by excluding those footholds which are in forbidden terrain cells and by eliminating footholds whose temporal kinematic margin is less than 0.4 second.

In the next chapter, the free gait algorithm will be evaluated on various terrain samples containing randomly placed obstacles. A timing analysis for each segment of the program will also be presented. In addition, the steep slope performance of the algorithm and its behavior in the presence of a leg failure will be evaluated.
CHAPTER 7

LISP Implementation of Free Gait Algorithm

7.1 Introduction

In this chapter, the LISP implementation of the free gait algorithm of Chapter 6 is described. After this description, a timing analysis is presented in order to find out which portions of the program are time-consuming tasks. This analysis is included because this program is intended for real-time application.

Following the above timing analysis, a statistical performance evaluation on terrain with randomly placed forbidden terrain cells is presented. The result of this evaluation demonstrates how effective this algorithm really is on terrain with obstacles. The performance evaluation of this algorithm on a slope is also included.

Finally, the behavior of the model vehicle in the presence of leg failure with the algorithm developed is discussed. For this purpose, three cases for one leg failure are explored because the six vehicle legs can be grouped into the front, middle, back legs by their functional characteristics. For the case of two leg failures, only the middle legs are disabled.

7.2 Overall Description of Program

The LISP implementation of the free gait algorithm in this dissertation closely follows the flowchart description of Chapter 6. Generally, one block of the flowchart is realized with one LISP function. Whenever a block in the flowchart is refined
into a detailed flowchart which contains several blocks, the LISP function is also hierarchically refined into several LISP functions, and the refined LISP functions are located in one level lower than that of the unrefined previous LISP function. Thus, the whole LISP program is represented with a tree-type structure, whose listing is included in Appendix B of this dissertation.

The top level function of this simulation program is named robot which realizes the flowchart of Fig. 6.4. This flowchart is composed of seven blocks, and the robot function is also realized by calling seven LISP functions, whose names are create-terrain, init-simulation, read-joystick, plan, execute, display, and increment-time. As described in the previous paragraph, the plan function, which is one of the seven functions, is implemented by calling one level lower functions, just as the plan block in the flowchart in Fig. 6.4 is refined into the flowchart in Fig. 6.5. In order to effectively manage these hierarchically refined function definitions, a tree structured file system is adopted by storing one function definition in one file during the program development. For example, the definitions of the seven functions used by the robot function are stored in the seven files in the [LISP.ROBOT] directory, but the robot function itself can be found in the [LISP] directory which is the highest level directory. This tree type file structure is constantly applied through the whole free gait program. In order to find the function definitions used in the plan function in the robot directory, the plan directory which is called [LISP.ROBOT.PLAN] should be checked. For example, the definition of the lift-exchange function used in the plan function is in the [LISP.ROBOT.PLAN] directory. Thus, the flowchart shown in Fig. 6.5 is implemented with the plan LISP function in the [LISP.ROBOT] directory. On the other hand, the flowchart in Fig. 6.6 is realized with the lift-exchange LISP function in the [LISP.ROBOT.PLAN] directory because this function is called by the former function. Therefore, in this
way, the hierarchy of the flowchart representation shown in Section 6.6 is preserved in the free gait motion coordination program written in LISP.

Sometimes, the above strict tree type structure during the program development cannot be maintained, because some functions already defined in other directories can be utilized again without re-defining them. For example, the place function in the [LISP.ROBOT.PLAN] directory uses four previously defined functions, which are not included in its sub-directory, [LISP.ROBOT.

PLAN.PLACE], but are in the [LISP.ROBOT.PLAIN], [LISP.ROBOT.PLAN.LIFT-EXC], and [LISP.ROBOT.PLAN.SEARCH-U] directories. This fact is explained inside the plan function definition with the comment showing the names of the external functions. Among the borrowed functions, the first function, change-state, is defined in the same directory in which the place function is. Thus, no directory name is given to the function name, but the other three function names are prefixed with the proper directory names. Though the above comment is added to the LISP function definitions for increasing their readability, they are optional to make them functions in a LISP environment.

In the LISP programming environment, the above mentioned tree type programming structure is actually encouraged during program development because a function can be easily defined and immediately tested with the incremental compilation facility. Practically, top-down design based on functional decomposition[53] is easily adopted in the LISP programming environment. Besides this advantage, the following Common LISP specific advantages are added in the free gait program.

Although the Flavors component of Zeta LISP is especially well suited to support object-oriented programming, the DOD (Department of Defense) standard LISP is Common LISP. Fortunately, the structure construct of Common LISP also supports object-oriented programming. Specifically, though Common LISP
structures are intended for data abstraction, if some entries of the fields in a structure are associated with procedures which are designed to modify some field contents, the structure can act like a template which can instantiate objects which possess status and behavior. In this program, the six legs are realized by utilizing the structure construct in the above way. Thus, to the free gait algorithm, the legs are considered as objects that possess all the necessary status information to represent the ASV legs and procedures to modify some of the status of the legs. For example, each leg object has its leg name, state, position, temporal kinematic margin, working volume, available foothold list, and temporal kinematic margin list associated with its available foothold list. Besides the above status information, the leg object knows how to change its current foot velocity control state to the next state, and how to move its leg from its current foot position to a next desired position. Specifically, the procedures defining the behavior of the leg object are given to it through the foot velocity control states, because the above two types of procedures, how to change its state and how to move its leg, can be associated with the control states themselves by utilizing the lambda expression of LISP which allows functions to be manipulated like data. Therefore, in order to update foot velocity states and foot positions, the free gait algorithm simply calls state-transition and move functions while passing a leg object as a parameter to each function. Upon a call, these functions send an appropriate message to the leg object by applying a corresponding function definition stored in the control states. Specifically, the how-to-change function is applied to change the foot velocity control states, and the how-to-move function is applied to modify the current position of a leg object.

In this program, the six leg objects are collected and made into a list called legs without adding any special relation among the leg objects such as priorities among
them. Therefore, the free gait algorithm can freely choose some legs from the leg list and manipulate them if necessary. Because of this implementation, there are two potential advantages. First, virtually any number of legs can be handled by this algorithm. In other words, this algorithm may be applied to a vehicle which has more or less than six legs, while the ASV, which is used as the model vehicle for this study, has six legs. This independence of the total number of legs of a vehicle provides a method for the vehicle to overcome a possible accident in which one of the vehicle legs is disabled during the operation of the vehicle. In the following section, this idea is actually investigated as one of the possible approaches to evaluate the performance of the algorithm developed. Second, different types of legs can be used in cooperation in this program because the program uses only abstracted information regarding leg objects, and details are handled by leg objects themselves. For example, though the geometries of the legs could be different from each other, all the legs can be equally handled by this program, if their foot velocity control state diagrams are common among the different legs.

In order to make this program run on a Symbolics 3640 LISP machine, it was necessary that the output of this simulation study be drawn in one of the windows displayed on the terminal of the Symbolics machine because a special graphic device was not available at the time. Unfortunately, on this machine, graphic facilities are not directly accessible from the Common LISP package, but indirectly accessible through the Zeta LISP package. Specifically, windows and the graphic facilities are implemented with Flavors on the machine, and Flavors can be utilized from the Zeta LISP package. Therefore, the Display Motion block of the free gait simulation program flowchart shown in Fig. 6.4 is written in Zeta LISP in order to utilize Flavors.

Because Flavors is implemented based on an object-oriented methodology,
actual drawing is performed by sending proper messages to the window called *
robot-window* in this program. Specifically, this window is made not to be exposed
on the screen, but rather to be utilized as a drawing buffer in order to eliminate any
flickering effect during the vehicle drawing. After the drawing process is finished,
the drawn image is copied to another window, named *
robot-display-window*, which
is actually exposed and visible on the screen. The copying process is performed
by the built-in high speed data copying facility called *bitblt* which is so fast that
the flickering is eliminated. In order to decrease the time required to draw one
simulation frame, the terrain is not repeatedly drawn in each cycle, but rather the
terrain data is copied using the *bitblt* from the terrain buffer which contains the
most recent terrain data made in the Create Terrain block of Fig. 6.4. This is
possible because the terrain data is fixed until starting a new simulation.

Differing from the graphic portion of this program, the other parts of the
simulation program are written in the Common LISP implemented in the LISP
machine. After all the necessary vehicle data are produced by the programs in
the Common LISP package, they are sent to the Zeta LISP package. Specifically,
in order to access a function in the Zeta LISP package from the Common LISP
package, the "package qualifier" *zl-user:* is attached in front of the name of the
function in the Zeta LISP package.

### 7.3 Timing Analysis

The simulation program of this dissertation is mainly written in Common
LISP while the simulation display part is realized with the Flavor system through
the Zeta LISP of the Symbolics LISP machine. In order to evaluate the language
choice, it is important to analyze the timing of the several sections of the program.
The front portions of the program, such as drawing terrain and initializing the
vehicle, are not time-critical because they are executed only once at the beginning. Thus, this part of the simulation program is not included in this timing analysis. Instead, only the plan, execute, and display portions of the whole program shown in Fig. 6.4 are analyzed in this section. The typical timing of these portions are shown in Table 7.1 which is measured from the Symbolics 3640 LISP machine without a floating point accelerator option board. The time entries of Table 7.1 are measured by the time function built in the LISP machine. This function returns two values. One is the amount of time spent by the LISP machine in order to execute the specific function. The other value is the amount of time the machine waits before it executes the function.

The first three lines of Table 7.1 show that almost the same amount of time is spent in each of the three blocks. The detailed timing of the Plan and the Execute blocks are also shown in Table 7.1. Though the plan function is made of eight functions, only seven functions are executed per each cycle because either the lift-exchange function or the place function is executed depending on the results of the stability test. This type of variation during the program execution is naturally expected almost any place where the branch type of decision is made.

The most significant effect of branching occurs in the execution of the search-update-TKM function which is a sub-function of the plan function. Specifically, the search-update-TKM function is composed of two functions as shown in the program listing in the appendix. The execution time of the first function, search-footholds, is influenced by the number of available legs, while that of the second part, update, is affected by the number of supporting legs. From observation of the program execution, it is found that the number of the supporting legs is usually three or four. Occasionally, but not often, it uses five or six legs in cases when the terrain is very difficult for the algorithm. On the other hand, the number
Table 7.1:
Timing analysis of the simulation program
Units: seconds

a) Typical Timing of Various Parts

<table>
<thead>
<tr>
<th></th>
<th>Time Spent</th>
<th>Waiting Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>0.227541</td>
<td>0.068</td>
</tr>
<tr>
<td>Execute</td>
<td>0.211211</td>
<td>0.004</td>
</tr>
<tr>
<td>Display</td>
<td>0.195252</td>
<td>0.007</td>
</tr>
</tbody>
</table>

b) Detailed Timing of Plan

<table>
<thead>
<tr>
<th></th>
<th>Time Spent</th>
<th>Waiting Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update-state-by-event</td>
<td>0.00758</td>
<td>0.000</td>
</tr>
<tr>
<td>Calculate-motion</td>
<td>0.13</td>
<td>0.008</td>
</tr>
<tr>
<td>Search-update-TKM</td>
<td>0.026425</td>
<td>0.006</td>
</tr>
<tr>
<td>Emergency-lift</td>
<td>0.000391</td>
<td>0.000</td>
</tr>
<tr>
<td>Stability test</td>
<td>0.005663</td>
<td>0.000</td>
</tr>
<tr>
<td>Lift-exchange/Place</td>
<td>0.000471/0.002646</td>
<td>0.000/0.000</td>
</tr>
<tr>
<td>Generate-command</td>
<td>0.000439</td>
<td>0.000</td>
</tr>
</tbody>
</table>
Table 7.1: continued...

c) Detailed Timing of *Execute*

<table>
<thead>
<tr>
<th></th>
<th>Time Spent</th>
<th>Waiting Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Leg-state-change</em></td>
<td>0.050782</td>
<td>0.001</td>
</tr>
<tr>
<td><em>Leg-move</em></td>
<td>0.199739</td>
<td>0.011</td>
</tr>
<tr>
<td><em>Check-event</em></td>
<td>0.003987</td>
<td>0.000</td>
</tr>
</tbody>
</table>

d) Two Timing Samples of *Search-footholds*

<table>
<thead>
<tr>
<th></th>
<th>First Sample</th>
<th>Second Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>With one leg</td>
<td>0.152639</td>
<td>0.266</td>
</tr>
<tr>
<td>With two legs</td>
<td>0.285039</td>
<td>0.4912</td>
</tr>
<tr>
<td>With three legs</td>
<td>0.409733</td>
<td>0.72</td>
</tr>
</tbody>
</table>
of available legs varies from zero to three almost freely. This is not because of
the terrain difficulty, but because of the temporal constraints imposed on the foot
trajectory. For example, if the terrain is totally permitted to the vehicle, only three
legs are usually used to support the body. In this case, the major limitation on leg
sequencing is the leg return time from the lift state to the ready state. In other
words, the supporting legs are ready to be lifted, but no legs are usually available.
As soon as one leg is available, the leg is exchanged with one of the supporting
legs. Therefore, the search function usually does not function except when one leg
is available to the vehicle.

The time required to execute the second function of the search-update function,
update-TKM, is also affected by the number of the legs in the supporting leg list.
From observation of the simulation results, the number of supporting legs is usually
three or four, and seldom five or six, as discussed before. However, the variation
of the execution time of the update-TKM function is not so great as that of the
first function.

From the above observation, it can be said that the search-foothold function
is not usually executed, but the update-TKM function is continuously executed
in each cycle. Therefore, the time required to execute the search-update-TKM
function, which is composed of the search-foothold and update-TKM function, is
usually equal to the time amount spent in the update-TKM function. In other
words, the measured time amount of the search-update-TKM function in Table
7.1 is actually equal to the time spent by the update-TKM function. However, if
the available-leg list has at least more than one leg, the timing is greatly changed.

Basically, the search-foothold function searches the all possible terrain cells
inside the working volume associated with the legs in the available leg list as
discussed in Section 6.2. The number of terrain cells searched by the algorithm
for a given leg depends on the number of forbidden cells included in its working volume, the terrain slope under the leg, and even the velocity of the vehicle body. However, the upper bound of the search for one leg is limited to 16 points. Thus, in the worst case, the 16 different temporal kinematic margins should be calculated to compare them and to pick one foothold for the given leg. Moreover, in order to calculate each TKM of the 16 possible footholds, almost the same function is called as the function used for the update-TKM function. Therefore, the search-foothold function requires roughly 10 times more time than the update-TKM if the same number of the legs is passed as their arguments. This timing ratio can be also observed from Table 7.1. Because it is observed that the vehicle has three supporting legs, when the time is measured, the time spent in order to execute the update-TKM function for one leg is roughly one-third of the time spent by the search-update-TKM function in the table, that is, roughly 0.009 seconds. Therefore, it can be expected that the time spent by the search-foothold for one leg will be around 0.14 seconds. From Table 7.1, it can be said that this expectation is quite reasonable.

If one leg is in the available leg list, the time spent by the plan is almost double the normal value since the time spent by the calculation-motion requires roughly the same amount of time as used by the search-update-TKM for one available leg because the time spent by the update-TKM is now negligible. If two legs or three legs are in the list, the time spent by the plan will be almost triple or quadruple, respectively. As explained before, the number of the legs in the available leg list seldom will be three. However, in the worst case, with three legs in the list, the Symbolics 3640 LISP machine without floating-point array board option will be slowed down to almost one second to complete one cycle of execution from the plan function to the display function. However, because the process to calculate
the temporal kinematic margin is mostly related to floating-point calculations, the execution speed is expected to be greatly improved with the floating point array option board, or by allowing this program to be run on a larger LISP machine such as the Symbolics 3675 LISP machine.

### 7.4 Statistical Evaluation of Performance on Terrain with Forbidden Cells

In order to evaluate the performance of the free gait algorithm developed in this dissertation, the model vehicle was made to walk over terrain containing randomly generated obstacles while it following a standard trajectory devised for this evaluation process. This standard trajectory is a straight line across the model terrain in the direction of the x coordinate of the earth coordinates. Along this trajectory, three types of motion are sequentially performed. Specifically, after the vehicle is initialized, it moves in the forward direction, makes a 90 degree turn-in-place at the middle of the terrain in the counter-clockwise direction, and then performs side stepping in the opposite direction of the y coordinate of the vehicle body until it reaches the end of the model terrain. If the vehicle is able to execute this entire maneuver without becoming deadlocked, this is counted as a successful motion. This standard trajectory is shown in Fig. 7.1. During maneuvering along the standard trajectory, the vehicle uses 50 percent of its maximum speed for all directions. (The maximum magnitudes of each velocity component commanded by a human operator are shown in Table 3.1.) For example, the vehicle is commanded with a one foot per second velocity input in the forward direction at the beginning of each evaluation. The other velocity components are shown in Fig. 7.1.

When a random terrain is generated, it is assumed that each cell has an independent probability to become a permitted or forbidden cell. In order to
Figure 7.1: Standard trajectory for the model vehicle for evaluation of performance in the presence of forbidden cells. (Velocity components of each segment of trajectory are shown at the end of the figure.)
simulate this condition, a simple random number generator is adopted using the
formula [59],

\[ x_{i+1} = (ax_i + b) \mod c \quad (7.1) \]

where \(a, b,\) and \(c\) are prime numbers. After \(x_{i+1}\) is generated by Eq. 7.1, it is
then divided by \(c\), and compared with a designated probability which is input by
an operator. If the value of \(x_{i+1}\) after normalization by \(c\) is less than the input
probability, a cell which is associated with it is assigned as a permitted cell. If not,
it will be a forbidden cell. A typical terrain example is shown in Fig. 7.2.

Performance tests were based on 10 trials on terrain with the same probability
of occurrence of permitted cells. Specifically, an operator inputs a different initial
random number as a seed for each trial, but the same probability for 10 trials.
This method permits new terrain to be generated for every trial, but with the
same probability of permitted cells for 10 trials. After the 10 experiments, a
percent successful motion is derived by dividing by 10 the number of successes in
the 10 trials. Results of this evaluation are shown in Fig. 7.3.

The general tendency of the results obtained is as would be expected. Specifi-
cally, the larger the probability of permitted cells is, the better is the chance that a
vehicle can cross the random terrain. With the free gait algorithm in this study, no
failures to complete the standard trajectory were observed for any terrain in which
permitted cells occupied 40 percent or more of the area of the whole simulation
terrain. If this percentage is lowered to 30, the observed chance of a successful
terrain transit is reduced to 90 percent. The success probability is further reduced
to 50 percent when only 20 percent of the area of the terrain contains permitted
cells. No successful motion was observed when 10 percent or less of the total
terrain cells are permitted. Therefore, practically, when 30 to 40 percent of the
Figure 7.2: Typical random terrain example
Figure 7.3: Experimental results for the algorithm on randomly generated terrain with zero-angle slope along the standard path.
total terrain cells are permitted, the free gait algorithm makes the ASV maneuver without becoming deadlocked. This performance is far superior to that of the only other free gait algorithm which has been subjected to a test of this type[11].

In order to obtain the result of Fig. 7.3, the safety margin of the free gait simulation program is set to 0.1 ft. When the safety margin is increased to 0.5 ft, the free gait performance on random terrain is shown in Fig. 7.4. As the margin is increased, the performance is slightly degraded. However, without great difficulty the free gait algorithm was able to make the vehicle cross random terrain in which permitted cells occupied 30 percent or more of the area of the whole simulation terrain.

When the safety margin is increased to 0.9 ft, the experimental results are as shown in Fig. 7.5. The performance is severely degraded compared with the previous experimental results. From observation, the most difficult maneuver of the vehicle driven by the free gait algorithm was the turn-in-place motion. Usually the vehicle was deadlocked during the turning motion. Therefore, it could be expected that if the vehicle had been allowed to move in a forward direction only, the performance would be greatly improved. In line with this expectation, the improved performance was actually observed from the experiments, and the result is shown in Fig. 7.6. When the vehicle used only forward motion, without great difficulty it could cross random terrain in which permitted cells occupied 40 percent or more of the area of the whole simulation terrain. In order to further evaluate the significance of the above results, an experiment was carried out in which the vehicle was made to walk on unobstructed flat terrain using a simple tripod gait with the same constrained walking volumes as used by the free gait algorithm. In this case, it was found that the minimal stability margin over any cycle of locomotion was 0.92 feet. Therefore, the 0.9 feet safety margin case represents a
Figure 7.4: Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when the safety margin is 0.5 feet.
Figure 7.5: Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when the safety margin is 0.9 feet.
Figure 7.6: Experimental results for the algorithm on randomly generated terrain with zero angle slope along a straight line when the safety margin is 0.9 feet.
very difficult test of is a fairly difficult margin for the free gait algorithm during locomotion on randomly generated terrain.

### 7.5 Steep Slope Performance

In order to evaluate the steep slope performance of the coordination algorithm developed, the same trajectory used in Section 7.4 was used. However, the right half of the terrain was tilted by a certain angle, while the left half was selected as flat terrain in order to allow the vehicle to initialize without any problem. For simplicity, the program developed is designed during its initialization to place the three vehicle legs from the assumption that the simulation terrain is flat. Following the same trajectory and velocity commands, the vehicle starts to walk, and then to climb up the hill of the simulation terrain. Because of the chosen trajectory, in the middle of the slope, the vehicle makes a 90 degree turn-in-place motion in the counterclockwise direction. Followed by this motion, the vehicle is made to start side stepping. Consequently, the trajectory tests one of the most difficult maneuvers that the vehicle can exhibit. Using the same type of evaluation criteria as in Section 7.4, it is determined whether or not the vehicle can manage terrain with a given slope.

From the simulation, it is observed that the algorithm makes the vehicle walk on inclined terrain which contains randomly placed obstacles with almost same ability shown on flat terrain regardless the slope of terrain when the angle slope is increased up to 10 degrees. However, the performance significantly drops if the angle is raised to 20 degrees. In this case, the vehicle seldom climbs up terrain with a slope even though the terrain does not contain any terrain obstacles. In other words, the performance on terrain with a slope seems to be not directly related to whether it contains many obstacles or not, but simply depends on the angle of the
terrain. This is evidently a weakness of the free gait algorithm of this dissertation which should be corrected in future research on stepping algorithm.

7.6 Behavior of Algorithm in Presence of Leg Failure

As with animals, occasional failure of one or more legs of a walking vehicle can be expected. In the presence of leg failure, the easiest approach to motion coordination is simply to make the vehicle stop. This can be achieved without great difficulty, as long as the vehicle speed is not high enough to prevent it from stopping immediately. Probably, this is the only choice for a vehicle operated with a predetermined leg sequencing algorithm, such as wave gaits. On the other hand, the free gait algorithm developed in this study provides an alternative way to overcome leg failure. Because the disabled leg is no longer useful to support or to propel the body, the leg can be eliminated from the legs list so that the algorithm may not use the leg any more. Therefore, for the gait algorithm, the vehicle effectively has five legs though it originally has six legs. This idea can be also applied to the case that two legs are disabled. That is, the two disabled legs can be eliminated from the legs list as if the vehicle had four legs. Naturally, the performance severely degrades when the number of disabled legs is increased. Theoretically, at least four legs should be included in the legs list because the algorithm uses at least three legs to maintain its static stability, and the remaining leg to change its supporting pattern (or to move its body).

In order to evaluate the algorithm performance on the ASV, the same standard trajectory and the same randomly generated terrain was used as in Section 7.4. From the simulation results, it was found that the performance of the vehicle significantly degrades. However, the important fact is that the vehicle is still operational with a disabled leg.
Specifically, when one of the front legs fails during vehicle operation, the free gait algorithm demonstrated the performance shown in Fig. 7.7 along the standard trajectory of Fig. 7.1. As discussed in Section 7.4, the turn-in-place motion was the most difficult maneuver for the vehicle operated by the free gait algorithm. When the vehicle used its forward motion only, it could cross the simulation terrain without deadlock condition when 80 percent or more of the area of the terrain was filled with permitted cells.

When one of the middle legs is inoperative, no successful motion along the standard trajectory was observed because the vehicle could not make a turn-in-place motion. This result is shown in Fig. 7.8. In other words, the middle legs are important to make the vehicle turn. Frequently, it was observed that the middle legs acted like pivots during turning. As the previous case, the vehicle could cross the simulation terrain in which 80 percent or more area of the terrain was composed with permitted cells when it used only its forward motion.

With one of the rear legs was inoperative, the performance shown in Fig. 7.9 was observed. Like the above two cases, the vehicle with one inoperative rear leg could overcome the same difficult random terrain that was crossed by the vehicle with one malfunctioning front or middle leg.

When the two middle legs of the six legs are disabled, the vehicle met frequent deadlock conditions after it traveled a very short distance. However, with help from a human operator, it can slowly walk again along a straight line. Specifically, when the vehicle is deadlocked, it can start to function again if the operator briefly gives a backward velocity command and then provides a forward velocity command. Consequently, the coordination algorithm is able to drive the vehicle with help from higher level intelligence when it loses two middle legs.
Figure 7.7: Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when one of the front legs is inoperative.
Figure 7.8: Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when one of the middle legs is inoperative.
Figure 7.9: Experimental results for the algorithm on randomly generated terrain with zero angle slope along the standard path when one of the rear legs is inoperative.
7.7 Summary

The free gait algorithm developed in this dissertation is implemented mainly in Common LISP, but in order to utilize a Symbolics 3640 LISP machine, the graphic part of the program is written in Zeta LISP and Flavors. Thus, except for the graphic part, the program can be hierarchically organized as the flowcharts of this dissertation are hierarchically arranged. Specifically, one block of the flowchart is realized with one LISP function. Therefore, the top LISP function, robot, calls several functions, and the functions called by the robot function again use some number of functions, as the top level flowchart is refined with several flowcharts in Section 6.6. This type of refinement is continued until a function called by the other functions can be implemented with a small number of LISP system built-in-functions. Because of this style, most of the LISP functions used in this program are usually realized with 10 to 20 lines of LISP code. Functions of this size can be easily debugged and directly read on the computer screen without moving the screen cursor. Naturally, the number of the functions is large in this program. The hierarchy among the functions greatly simplifies the task of handling the large number of functions in a systematic way.

In the evaluation process performed in this chapter, no human intervention is involved to drive the vehicle on the test terrain because a standard trajectory is chosen. However, in an actual application of the free gait algorithm, an operator will guide the direction of the movement of the vehicle. This action can be interpreted as a higher level terrain selection. Therefore, this action will help the mobility of a vehicle. For example, if a vehicle enters the deadlock condition, then the operator may make it escape from the deadlock status by providing an opposite directional command. This fact implies that even though a vehicle cannot cross
a terrain with the standard trajectory, human intervention can make it cross the same terrain successfully. From this consideration and the evaluation results, this free gait algorithm can be applied to terrain in which up to 70 percent area of the whole terrain is filled with forbidden terrain cells without difficulty, and if human intervention is allowed, it is expected that this algorithm can be applied to even more difficult terrain.

If one of the six legs is disabled during its operation, the vehicle can automatically adapt to this malfunction with the free gait algorithm developed in this dissertation, though its maneuverability is degraded. Without one leg, the vehicle virtually is operational without great difficulty. Usually, the commanded velocity is not achieved because of the deceleration method which is activated whenever the vehicle meets unfavorable conditions. When the middle two legs are disabled, the vehicle can follow only very slow speed commands, but it can move to the destination with a human operator's intelligence which can help the vehicle to get out from the deadlock condition. Consequently, when one of the legs is disabled, the algorithm practically overcomes the problem, but when two legs are malfunctioning, the algorithm can function only with help from a human operator. However, the ability of the vehicle to survive with leg failure also depends on its geometry.
CHAPTER 8

Summary and Conclusions

In this dissertation, an omni-directionally controllable free gait motion coordination algorithm for a six legged walking machine over three-dimensional rough-terrain is developed and evaluated by means of a simulation study. The existing Adaptive Suspension Vehicle is chosen as a vehicle model, and prismatic terrain as the terrain model. With the motion coordination algorithm developed, the vehicle can select suitable footholds along a path commanded by an operator. In these procedures, a specified minimum stability margin is automatically maintained because the algorithm always tries to maximize the vehicle stability margin and the leg temporal kinematic margins. The value of this minimum margin is also assigned by an operator. However, if the vehicle cannot follow the velocity commands, the coordination algorithm actively resists the commands until the unfavorable situation is resolved. As a result, the algorithm developed is the first complete motion coordination algorithm which explicitly optimizes leg temporal kinematic margins as a primary criterion while satisfying a stability constraint during rough-terrain locomotion utilizing free gaits. Moreover, when possible, stability is also optimized as a secondary criterion.

165
8.1 Research Contributions

The main strategy of the free gait algorithm of this dissertation is to maximize the number of legs in the air so that these legs can be utilized when the vehicle mobility is limited either because of a small stability margin of the vehicle body or because of a small temporal kinematic margin of one of the vehicle supporting legs. Specifically, in this dissertation, the test for the use of an additional leg is expanded from only the stability condition to a unfavorable condition which includes the small stability condition and the small temporal kinematic condition. In order to help the vehicle to recover from the unfavorable condition, the algorithm developed uses a body deceleration method. This scheme is also activated so that the algorithm can solve the condition without concern for a time limit because the vehicle speed slowed down by the scheme delays the occurrence of the condition.

Until now, the implementation of free gaits has required an assumption of either weight-less legs [10] or the use of a time-delay queue to compensate for the time difference between the planned motion and the presently executed motion [11]. The first approach is almost impossible to implement in a real walking machine, and the second approach is not convenient for practical use because the vehicle response is effective one second after the time when the operator changes the vehicle velocity commands. In this study, the time delay queue is eliminated, and a new finite-state machine, the six state leg motion planning state machine, takes care of the time difference between the planned motion and the presently executed motion. This approach relaxes the very tight time restriction of the previous work reported in [11]. Specifically, because the algorithm always projects one second into the future, the projected leg events are tracked and controlled by the six leg motion planning state machines. Thus, these machines automatically
translate future decisions from the planning algorithm to present leg commands which can be directly given to their foot velocity control finite-state machines. Therefore, this control structure acts as a bridge between the planning algorithm and the foot velocity control finite-state machines. Small timing errors practically expected are automatically re-synchronized by this sequential machine inside the program. Consequently, the velocity commands from an operator can immediately affect the vehicle body motion planning.

The above described hierarchy relates to the low-level motion coordination structure. The high level planning algorithm is also hierarchically organized. Therefore, the whole program is hierarchically organized. This control structure is well suited to deal with the logical complexity of this coordination algorithm, which is much higher than any control algorithm using periodic gaits. This structure provides an algorithm developer with a much easier way to modify, test, and document an algorithm during research. The same structure is preserved in the LISP implementation, and this method is very effective in the implementation process. The development cost is thereby significantly reduced.

The foothold selection process developed adopts the problem solving paradigm known as a generate-and-test system [52]. In this program, the generator which is complete, nonredundant, and informed, tries all the available footholds one after another in the order of their temporal kinematic margins while the footholds are already limited by excluding those footholds which are in forbidden terrain cells and by eliminating footholds whose temporal kinematic margin is less than 0.4 second because these footholds are not reachable when the vehicle actually tries to use them as supporting points.

Legs are implemented with an object-oriented programming style. Thus, the coordination algorithm does not have to explicitly control the state transitions of
the states of the six foot velocity control state machines, but it can handle them in a uniform way to make a state transition without being concerned with details. Because of this approach, two potential advantages are introduced. First, the free gait algorithm can handle any number of legs without modifying the algorithm. This independence from the total number of legs provides a means to overcome a leg failure by eliminating the disabled leg. Second, different types of legs can be incorporated as long as their abstracted characteristics are consistent with the algorithm.

Basically, the free gait algorithm represented is designed to handle one leg placement and one leg lift per one cycle time for simplicity. Sometimes, this strategy is not adequate to solve an unfavorable condition resulting from the lack of vehicle stability and/or inadequate temporal kinematic margins, but can be solved by using two or three legs. In order to simulate this problem, whenever the basic free gait algorithm fails to find one appropriate leg to solve the condition, one leg is forced to support the vehicle considering only the kinematic margin of the legs. This leg acts like a seed to find another proper leg by the basic free gait algorithm which optimizes both the vehicle stability margin and the temporal kinematic margin of the leg. Therefore, a pseudo-multiple-leg solution is found in certain situations.

8.2 Research Extensions

In order to improve the performance of the free gait algorithm developed, the deadlock recovery portion of the algorithm should be improved. When the vehicle is deadlocked, the center of the vehicle body cannot be moved any further, but as long as the vertical projection of the center is not moved, the attitude or the altitude of the vehicle body can be changed. If this is possible, the relative position
between the working volume associated with the legs and the supporting terrain is changed. In other words, new possible supporting positions on the terrain are provided to the vehicle. Thus, new chances are given to the vehicle so that it can get out from the deadlock condition. However, the way to change the attitude or the altitude of the body is subject to further study.

The body deceleration method developed in this dissertation is based on a simple deceleration factor which is increased whenever the vehicle meets unfavorable conditions. This simple scheme works well in simulation form, but when it is applied to the real vehicle, it should be refined or replaced with a more sophisticated method so that the effects of the center of pressure [22] of the vehicle, and the body acceleration can contribute to the deceleration method. This simple body deceleration method is believed to be one of the reasons the performance of the algorithm on steep slope degrades so seriously.

The simulation study reported herein was of a quasi-static nature in the sense that all vehicle acceleration forces were ignored. Thus, the values obtained for stability margin are only approximate. Real stability depends upon the relationship between the center of pressure and the forces of the supporting legs, not just the vertical projection of the center of gravity. Future simulation studies should take this factor into account by explicitly modeling the effects of limb and body accelerations.

Finally, in order to improve the algorithm presented, a new approach may be appropriate because the complexity of the algorithm will grow. Though the algorithm is hierarchically organized, future improvement, modifying and verifying will present more difficulties than the current situation if all the control of logic flow is restricted to the conventional explicit style. Therefore, the coordination algorithm presented may be more easily improved if the logic flow is replaced
with a logic programming style or a rule-base system because the explicit logic control or computation can be considered as a special case of logical deduction, with aids of the object-oriented style already utilized in the program developed. Thus, a multi-paradigm programming environment such as provided by various commercial expert system shells may be more suitable for continuation of this work, at least for the development of prototype software. Ultimately, it is to be hoped that a field-testable system utilizing this approach could also be developed.

The above mentioned future research extensions are derived from problems encountered during this algorithm development. Hopefully, the suggested future extensions will lead to more efficient and realistic free gait algorithms for rough-terrain locomotion of walking machines.
APPENDICES
Computer Programs

The appendices list the computer programs used in the course of this research, and are organized into two parts. The first part includes the program listing discussed in Chapter 4, and the second part lists the whole simulation program.

APPENDIX A. Program Listings Referred to in Chapter 4
1. Pascal Implementation
2. LISP Implementation
3. Prolog Implementation
4. OPS5 Implementation
5. Smalltalk Implementation

APPENDIX B. Program Listings of Free Gait Motion Coordination Algorithm
1. Free Gait Motion Coordination Algorithm
2. Graphic Primitives
3. Data File for Simulation Vehicle Model
APPENDIX A

Program Listings Referred to in Chapter 4
A.1 Pascal Implementation of
Foot Velocity Control Finite State Machine
program leg(input, output);

const t1=0.6;
t2=0.999999; { 1 second }
t3=0.4;
t4=0.6;
type kinds_of_command = (deploy_command, recover_command);
kinds_of_event = (contact_confirm);
kinds_of_states = (ready, advance, descent, contact, support, lift, return);
commands = set of kinds_of_command;
events = set of kinds_of_event;

var leg_command : commands;
leg_event : events;
leg_state : kinds_of_states;
end_time, time, leg_clock : real;

begin
  leg_command := [ deploy_command, recover_command];
  leg_event := [ contact_confirm ];
  writeln( 'Please input end time');
  readln(end_time);

  leg_state := ready;
time := 0;
leg_clock :=0;

repeat
  case leg_state of
    ready : if deploy_command in leg_command then
      begin
        leg_state := advance;
        leg_clock := 0
      end;
    advance : if leg_clock >= t1 then
      leg_state := descent;
    descent : if contact_confirm in leg_event then
      begin
        leg_state := contact;
        leg_clock := 0
      end;
  end;

end;
contact : if leg_clock >= t2 then
    leg_state := support;

support : if recover_command in leg_command then
    begin
    leg_state := lift;
    leg_clock := 0
    end;

lift : if leg_clock >= t3 then
    begin
    leg_state := return;
    leg_clock := 0
    end;

return : if leg_clock >= t4 then
    leg_state := ready;

end; { case leg_state of }

time := time + 0.1;
leg_clock := leg_clock + 0.1;

write('Time is ',time,' State is ');
case leg_state of
    ready : writeln('Ready');
    advance : writeln('Advance');
    descent : writeln('Descent');
    contact : writeln('Contact');
    support : writeln('Support');
    lift : writeln('Lift');
    return : writeln('Return');
end;

until end_time <= time;
end.
A.2 LISP Implementation of
Foot Velocity Control Finite State Machine
(defun leg (end-time)

; Initialization
(setf t1 0.6  t2 0.999999 t3 0.4  t4 0.6  state 'ready)

; Following list are defined for simulation.
(setf command '(deploy recover)  event '(contact))
; from other levels

; Start state transition.
(do ((time 0 (+ time 0.1))
     (leg-clock 0 (+ leg-clock 0.1)))
   ((< end-time time) 'done)

   (cond ((and (equal state 'ready) (equal 'deploy (car command)))
          (setf state 'advance)
          (setf leg-clock 0))

   ((and (equal state 'advance) (>= leg-clock t1))
    (setf state 'descent))

   ((and (equal state 'descent) (equal 'contact (car event)))
    (setf state 'contact)
    (setf leg-clock 0))

   ((and (equal state 'contact) (>= leg-clock t2))
    (setf state 'support))

   ((and (equal state 'support) (equal 'recover (cadr command)))
    (setf state 'lift)
    (setf leg-clock 0))

   ((and (equal state 'lift) (>= leg-clock t3))
    (setf state 'return)
    (setf leg-clock 0))

   ((and (equal state 'return) (>= leg-clock t4))
    (setf state 'ready))
   (T))

   (setf show-time (list 'Time 'is time))
   (setf show-state (list 'State 'is state))
   (print show-time) (princ " ") (prinl show-state)
)

178
A.3 Prolog Implementation of
Foot Velocity Control Finite State Machine
t1(0.599999). /* 0.6 sec */
t2(0.999999). /* 1.0 sec */
t3(0.399999). /* 0.4 sec */
t4(0.599999). /* 0.6 sec */
dt(0.1).

time(0).
leg_clk(0).
state(ready).
command([deploy|recover]).
event([contact]).

leg(Final_time) :- repeat, cycle, time(T), T >= Final_time.
cycle :- update_time(T1),
update_leg_clk,
state(X),
new_state(Y), retract(state(X)), asserta(state(Y)),
clear_leg_clk(X,Y),
write([time,is,T1,state,is,X]),nl,!

update_time(T1) :- time(T1), dt(Dt), T is T1 + Dt,
retract(time(T1)), asserta(time(T)).

update_leg_clk :- leg_clk(T2), dt(Dt), T3 is T2 + Dt,
retract(leg_clk(T2)), asserta(leg_clk(T3)).
clear_leg_clk(X,Y) :- X \= Y,
retract(leg_clk(T4)), asserta(leg_clk(0)).
clear_leg_clk(X,Y).

new_state(advance) :- state(ready), command([deploy|_]).
new_state(descent) :- state(advance), leg_clk(T1), t1(T), T >= T1.
new_state(contact) :- state(descent), event([contact]).
new_state(support) :- state(contact), leg_clk(T2), t2(T), T >= T2.
new_state(lift) :- state(support), command([_|recover]).
new_state(return) :- state(lift), leg_clk(T3), t3(T), T >= T3.
new_state(ready) :- state(return), leg_clk(T4), t4(T), T >= T4.
new_state(X) :- state(X).
A.4 OPS5 Implementation of
Foot Velocity Control Finite State Machine
(literalize time_constant  t1 t2 t3 t4 dt)
(literalize time  t)
(literalize leg_state  state)
(literalize leg_clock  t)
(literalize leg_command  command)
(literalize leg_event  event)
(literalize final_time  t)
(vector-attribute command)
(vector-attribute event)

(p start1
  (start 1)
  -->
  (make action  change_time)
  (make time_constant
    "t1 0.6  t2 1.0  t3 0.4  t4 0.6  dt 0.1")
  (make time  "t 0.0")
  (make leg_state  "state advance")
  (make leg_clock  "t 0.0")
  (make leg_command  "command deploy recover")
  (make leg_event  "event contact")
  (write (crlf) Please input final time)
  (make final_time  "t (accept)"
    (write (crlf) Initialization is done))

(p increase_time
  (action  change_time)
  (time  "t <time>")
  (leg_clock  "t <clock>")
  (time_constant  "dt <dt>")
  (leg_state  "state <state>")
  (final_time  "t >= <time>")
  -->
  (modify 2  "t (compute <time> + <dt>)")
  (modify 3  "t (compute <clock> + <dt>)")
  (remove 1)
  (make action  change_state)
  (write (crlf)
    state: <state> leg_clock: <clock> time: <time>)

(p change_to_advance
  (action  change_state)
  (leg_state  "state ready")
  (leg_clock  "t <clock>")
  (leg_command  "command deploy")
  -->
  (modify 2  "state advance")
(modify 3 ~t 0.0)
(make action change_time))

(p change_to_descent
  (action change_state)
  (leg_state ~state advance)
  (leg_clock ~t <clock>)
  (time_constant ~t1 <= <clock>)
  --->
  (modify 2 ~state descent)
  (remove 1)
  (make action change_time))

(p change_to_contact
  (action change_state)
  (leg_state ~state descent)
  (leg_event ~event contact)
  (leg_clock ~t <clock>)
  --->
  (modify 2 ~state contact)
  (remove 1)
  (make action change_time)
  (modify 4 ~t 0.0))

(p change_to_support
  (action change_state)
  (leg_state ~state contact)
  (leg_clock ~t <clock>)
  (time_constant ~t2 <= <clock>)
  --->
  (modify 2 ~state support)
  (remove 1)
  (make action change_time))

(p change_to_lift
  (action change_state)
  (leg_state ~state support)
  (leg_command ~command <x> recover)
  (leg_clock ~t <clock>)
  --->
  (modify 2 ~state lift)
  (remove 1)
  (make action change_time)
  (modify 4 ~t 0.0))
(p change_to_return
  (action change_state)
  (leg_state ~state lift)
  (leg_clock ~t <clock>)
  (time_constant ~t3 <= ~clock>)
  -->
  (modify 2 ~state return)
  (modify 3 ~t 0.0)
  (remove 1)
  (make action change_time))

(p change_to_ready
  (action change_state)
  (leg_state ~state return)
  (leg_clock ~t <clock>)
  (time_constant ~t4 <= <clock>)
  -->
  (modify 2 ~state ready)
  (remove 1)
  (make action change_time))

(p switch_control
  (action change_state)
  -->
  (remove 1)
  (make action change_time))
A.5 Smalltalk Implementation of
Foot Velocity Control Finite State Machine
Object subclass: #LegExecutor
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

LegExecutor class methods

LegExecutor methods

execute: aLeg until: endTime
  "Execute a leg until endTime."
  | globalTime |
  globalTime := 0.0 .
  ((aLeg getState) getName) printOn: Transcript.
      Transcript cr.
  [ globalTime <= endTime ]
  whileTrue: [
    globalTime := globalTime + 0.1.
    aLeg changeState.
    ((aLeg getState) getName) printOn: Transcript.
      Transcript cr.]
Object subclass: #Leg

instanceVariableNames:
  'legState events ready advance descent contact support
   lift return samplingTime'

classVariableNames: '

poolDictionaries: ''

Leg class methods

Leg methods

changeState

"Change state by event or time out."

legState :=
  (#( Ready Descent Support ) includes: (legState getName))
  ifTrue: [ legState doTransition:
    ( events at: ( legState getName ) ) ]
  ifFalse: [ legState doTransition: samplingTime ].

getState

"Return State of a leg."

^legState.

initialize

"Initialize leg to Ready state."

| t1 t2 t3 t4 |
|----|----|----|----|
| 0.6 |
| 0.999999. " means 1.0 " |
| 0.4 |
| 0.6 |

samplingTime := 0.1.

events := Dictionary new.

events at: #Ready put: #DeployCommand.

events at: #Descent put: #ContactConfirm.

events at: #Support put: #LiftCommand.

ready := AsyncState new.

ready setName: #Ready.

ready setEvent: #DeployCommand.

advance := SyncState new.

advance setName: #Advance.

advance setTimeout: t1.

descent := AsyncState new.

descent setName: #Descent.

descent setEvent: #ContactConfirm.

contact := SyncState new.

contact setName: #Contact.

contact setTimeout: t2.

support := AsyncState new.

support setName: #Support.
support setEvent: #LiftCommand.

lift := SyncState new.
lift setName: #Lift.
lift setTimeout: t3.

return := SyncState new.
return setName: #Return.
return setTimeout: t4.

ready setState: advance.
advance setState: descent.
descent setState: contact.
contact setState: support.
support setState: lift.
lift setState: return.
return setState: ready.

legState := ready.
Object subclass: #State
  instanceVariableNames:
    'name nextState '
  classVariableNames: ''
  poolDictionaries: ''

State class methods

State methods

getName
  "Returns a state name"
  ^name.

getNextState
  "Returns a next state"
  ^nextState.

setName: stateName
  "Set an state name into a state"
  name := stateName.

setNextState: stateName
  "Set next state name"
  nextState := stateName.
State subclass: #AsyncState
  instanceVariableNames:
    'event '
  classVariableNames: '
  poolDictionaries: '

AsyncState class methods

AsyncState methods

doTransition: condition
  "Do transition to the next state if condition
  is equal to the termination event"
  condition = event
    ifTrue: [ nextState initialize. ^nextState ]
    ifFalse: [ ^self ].

getEvent
  "Return terminating event of an asynchronous state"
  ^event.

initialize
  "Initialize a state. This is dummy but useful for
  synchronous state."
  ^nil.

setEvent: eventName
  "Set event name of the state"
  event := eventName.
State subclass: #SyncState
  
  instanceVariableNames:
i    'timeOut time '
  
  classVariableNames: ''
  
  poolDictionaries: ''

SyncState class methods

SyncState methods

doTransition: deltaTime
  "Make transition if time is equal to or greater than
time out constant, and return proper state."
  
time := time + deltaTime.
  
  time >= timeOut
 .isTrue: [ nextState initialize. nextState ]
  .false: [ .self ].

getTime
  "Return time of a state."
  time.

getTimeOut
  "Return time out constant."
  timeOut.

initialize
  "Initialize synchronous state. (Reset time.)"
  time := 0.0 .

setTime: aTime
  "Set time to aTime."
  time := aTime.

setTimeOut: timeAmount
  "Set time out constant"
  timeOut := timeAmount.
APPENDIX B

Program Listings of Free Gait Motion Coordination Algorithm
B.1 Free Gait Motion Coordination Algorithm
(defvar additional-TKM-margin-in-place)
(defvar available-legs)
(defvar body-rotate-rate)
(defvar body-rotate-rate1)
(defvar body-rotate-rate6)
(defvar body-rotate-rate10)
(defvar body-trans-rate)
(defvar body-trans-rate1)
(defvar body-trans-rate6)
(defvar body-trans-rate10)
(defvar command)
(defvar contact-state-legs)
(defvar convex-hull-order)
(defvar deadlock)
(defvar deceleration-factor)
(defvar deleted-leg)
(defvar deploy-flag)
(defvar eligible-to-lift-legs)
(defvar estimated-support-plane)
(defvar estimated-support-plane-wrt-body)
(defvar eta1)
(defvar eta2)
(defvar H)
(defvar H1)
(defvar H6)
(defvar H10)
(defvar inv-H)
(defvar inv-H1)
(defvar inv-H6)
(defvar inv-H10)
(defvar largest-stability-leg)
(defvar legs)
(defvar leg1)
(defvar leg2)
(defvar leg3)
(defvar leg4)
(defvar leg5)
(defvar leg6)
(defvar max-eta)
(defvar max-height)
(defvar min-eta)
(defvar min-height)
(defvar planned-contact-time)
(defvar present-joystick-command)
(defvar projected-time)
(defvar ready-state-legs)
(defvar recover-flag)
(defvar safety-margin)
(defvar sampling-time)
(defvar supporting-legs)
(defvar support-state-legs)
(defvar T1)
(defvar T2)
(defvar T3)
(defvar T4)
(defvar terrain)
(defvar terrain-height)
(defvar tkm-margin)
(defvar zero-time)

(defvar vehicle-points (make-array 28))
(defvar vehicle-points-earth (make-array 28))
(defvar body-points (make-array 10))
(defvar previous-vehicle-points nil)
(defvar polygons (make-array 13))
(defvar numpolys nil)
(defvar vertices (make-array 100))
(defvar eye-space nil)
(defvar middle-of-screen nil)
(defvar terrain-data (make-array '(39 39) :initial-element 0))
(defvar terrain-height-list)
(defvar terrain-height-array)
(defvar cursor-x 19) (defvar cursor-y 19)
(defvar joy-x 0)
(defvar joy-y 0)
(defvar joy-r 0)

(defstruct (leg)
    (name nil)
    (state nil)
    (time 0.0)
    (TKM 0.0)
    (pos nil)
    (desired-pos nil)
(contact-confirm nil)
(projected-state nil)
(projected-time 0.0)
(projected-TKM 0.0)
(projected-pos nil)
(projected-permitted-footholds nil)
(projected-TKM-list nil)
(no-cells-available-flag)
(sixteen-footholds nil)
(working-volume nil)
(four-lines nil)
(ready-position nil)
(exchanged-name nil)
)

(defun init-available-legs ()
  (setf available-legs nil))

(defun init-command()
  (setf (get 'command 'deploy) nil) ; field deploy can have only one leg
  (setf (get 'command 'recover) nil)); field recover can a list of legs

(defun init-contact-state-legs ()
  (setf contact-state-legs nil))

(defun init-deadlock ()
  (setf deadlock nil))

(defun init-deceleration-factor ()
  (setf deceleration-factor 0))

(defun init-deleted-leg ()
  (setf deleted-leg nil))

(defun init-deploy-flag ()

196
(setf deploy-flag nil))

(defun init-eligible-to-lift-legs ()
  (setf eligible-to-lift-legs nil))

(defun init-Hs ()
  ; library function : ident
  (setf H I (ident))
  (setf (aref H I 0 3) 6.5)
  (setf (aref H I 1 3) 19.5)
  (setf (aref H I 2 3) 5.4)
  (setf H6 H1)
  (setf H10 H1))

(defun init-legs ()
  (let* ((x (aref H I 0 3))
          (y (aref H I 1 3))
          (z (aref H I 2 3)))
    (setf leg1 (make-leg
                 :name 'leg1
                 :sixteen-footholds (get 'leg-1 'sixteen-footholds)
                 :four-lines (get 'leg-1 'four-lines)
                 :working-volume (get 'leg-1 'working-volume)
                 :ready-position (get 'leg-1 'ready-position)
                 :projected-pos (list (+ x 5) (+ y 3) 0)
                 :projected-state 'eligible-to-lift
                 :pos (list 5 3 (- 0 z))
                 :state 'support))
    (setf leg2 (make-leg
                 :name 'leg2
                 :sixteen-footholds (get 'leg-2 'sixteen-footholds)
                 :four-lines (get 'leg-2 'four-lines)
                 :working-volume (get 'leg-2 'working-volume)
                 :ready-position (get 'leg-2 'ready-position)
                 :projected-pos (list (+ x 5) (- y 3) 0) ; dummy
                 :projected-state 'available-leg
                 :pos (get 'leg-2 'ready-position)
                 :state 'ready))
    (setf leg3 (make-leg
                 :name 'leg3
                 :sixteen-footholds (get 'leg-3 'sixteen-footholds)
                 :four-lines (get 'leg-3 'four-lines)
                 :working-volume (get 'leg-3 'working-volume)
                 :ready-position (get 'leg-3 'ready-position)
                 :projected-pos (list x (+ y 3) 0) ; dummy
                 :pos (list 5 3 (- 0 z))
                 :state 'support))

197
(defun init-present-joystick-command ()
  (setf present-joystick-command '(0 0 0)))

(defun init-ready-state-legs ()
  (setf ready-state-legs nil))

(defun init-recover-flag ()
  (setf recover-flag nil))

198
(defun init-body-rotate-rates ()
  (setf body-rotate-rate1 '(0 0 0))
  (setf body-rotate-rate6 '(0 0 0))
  (setf body-rotate-rate10 '(0 0 0)))

(defun init-supporting-legs ()
  (setf supporting-legs nil))

(defun init-support-state-legs ()
  (setf support-state-legs nil))

(defun init-terrain ()
  (setf terrain (make-array '(39 39) :initial-element 0))
  (setf terrain-height (make-array '(39) :initial-element 0)))

(defun init-body-trans-rates ()
  (setf body-trans-rate1 '(0 0 0))
  (setf body-trans-rate6 '(0 0 0))
  (setf body-trans-rate10 '(0 0 0)))

(defun read-terrain ()
  (dotimes (i 39)
    (setf (aref terrain-height i) (aref terrain-height-array i))
    (print safety-margin)
    (print "safety-margin")
    (setf safety-margin (read))
    (dotimes (i (first (array-dimensions terrain-data)))
      (dotimes (j (second (array-dimensions terrain-data)))
        (setf (aref terrain i j) (aref terrain-data i j)))))

(defun set-four-lines ()
  (setf (get 'leg-1 '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)))))

(defun set-four-lines ()
  (setf (get 'leg-2 '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))))
(defun set-ready-positions ()
  (setf (get 'leg-1 'ready-position) '( 5 3 -4))
  (setf (get 'leg-2 'ready-position) '( 5 -3 -4))
  (setf (get 'leg-3 'ready-position) '( 0 3 -4))
  (setf (get 'leg-4 'ready-position) '( 0 -3 -4))
  (setf (get 'leg-5 'ready-position) '(-6 3 -4))
  (setf (get 'leg-6 ready-position) '(-6 -3 -4)))

(defun set-sixteen-footholds ()
  (setf (get 'leg-1 '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 (get 'leg-2 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 (get 'leg-3 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)
          ( 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)
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 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
          ( 1.5 -4.3) ( 1.5 -3.3) ( 1.5 -2.3) ( 1.5 -1.3)
(setf (get 'leg-4 '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 (get 'leg-5 '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 (get 'leg-6 '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))))

(defun set-working-volumes ()
  (setf (get 'leg-1 '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))
  (setf (get 'leg-2 '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))
  (setf (get 'leg-3 '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))
  (setf (get 'leg-4 '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))
  (setf (get 'leg-5 '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))
  (setf (get 'leg-6 '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))))
(defun robot()
  (create-terrain)
  (init-simulation)
  (do ()
    (nil) ; forever
    (increase-time)
    (read-joystick)
    (plan)
    (execute)
    (display)))
(defun create-terrain()
  (init-vehicle-globals)
  (zl-user:make-robot-window)
  (setf middle-of-screen
        (list (/ (zl-user:send zl-user:*robot-window* :inside-width) 2)
              (/ (zl-user:send zl-user:*robot-window* :inside-height) 2)))
  (setf eye-space (eye-trans (list 500 0 0)))
  (read-terrain-height)
  (draw-terrain eye-space)
  (display-cursor)
  (zl-user:make-visible)
  (print "Do you want to continue? Hit a key and return key")
  (read)
  (erase-obstacles)
  (do ((radius 500) (alpha 0) (beta 0) (delta 0.0001)
        (joystick-value nil)
        (end-flag nil))
      (end-flag (save-terrain))
    (zl-user:make-visible)
    (setf joystick-value (read-simulated-joystick))
    (let ((x (first joystick-value))
           (y (second joystick-value))
           (r (third joystick-value))
           (fire (fourth joystick-value)))
      (if eye-space
          (erase-terrain eye-space))
      (cond
        (fire (setf end-flag t))
        ((> x delta) (setf alpha (+ alpha 0.1)))
        ((< x (- delta)) (setf alpha (- alpha 0.1)))
        ((> y delta) (setf beta (+ beta 0.1)))
        ((< y (- delta)) (setf beta (- beta 0.1)))
        ((> r delta) (setf radius (+ radius 10)))
        ((< r (- delta)) (setf radius (- radius 10)))
        (setf eye-space (eye-trans (list radius alpha beta)))
        (draw-terrain eye-space))))
(defun displayQ()
  (let ((out-legs (list leg1 leg2 leg3 leg4 leg5 leg6)))
    (do ((legs out-legs (cdr legs))
      (foot-positions nil))
      ((null legs)
        (display-vehicle H1 (reverse foot-positions))
        (zl-user:copy-terrain-to-robot-window)
        (zl-user:make-visible))
      (setf foot-positions
        (cons (leg-pos (car legs)) foot-positions))))

(defun execute()
  (leg-state-change legs)
  (leg-move legs) ; output to display
  (check-event legs)
  'done-execute)

(defun increase-time()
  (dolist (a-leg legs)
    (setf (leg-time a-leg)
      (+ sampling-time (leg-time a-leg)))))

(defun init-simulation ()
  (setf safety-margin (/ 1 12))
  (setf TKM-margin 0.4)
  (setf additional-TKM-margin-in-place 0.2)
  (setf projected-time 1.0)
  (setf T1 0.6 T2 1.0 T3 0.4 T4 0.6)
  (setf planned-contact-time 0.4)
  (setf sampling-time 0.1)
  (setf zero-time 0.0)
  (setf convex-hull-order '(leg2 leg4 leg6 leg5 leg3 leg1))

; body-regulation constants
(setf min-eta 0.0000001) ; 0 degree
(setf max-eta 0.3491) ; 20 degrees
(setf min-height 4.0) ; 4.4 feet
(setf max-height 5.4) ; 5.4 feet
(setf eta1 min-eta) ; 0 degree
(setf eta2 0.5236) ; 30 degree

(set-four-lines)
(set-ready-positions)
(set-sixteen-footholds)
(set-working-volumes)
(init-deceleration-factor)
(init-present-joystick-command)
(init-terrain)
(init-Hs)
(init-body-rotate-rates)
(init-body-trans-rates)

(init-legs)
(setf legs (list leg1 leg2 leg3 leg4 leg5 leg6))
(init-available-legs)
(init-eligible-to-lift-legs)
(init-supporting-legs)
(init-contact-state-legs)
(init-ready-state-legs)
(init-support-state-legs)
(init-deploy-flag)
(init-recover-flag)
(init-deleted-leg)
(init-deadlock)
(init-command)
(read-terrain)

(defun plan()
  (update-state-by-event)
  (calculate-motion) ; calculate body motion
  (search-update-TKM) ; search footholds for available legs
  (emergency-lift)
  (if (stability safety-margin)
      (lift-exchange) ; search for leg lift or exchange
            (place)) ; else search for additional supporting leg
  (generate-command) ; generate leg-command
  'done-plan)

(defun read-joystick()
  (do ((exit-flag nil))
    (exit-flag 'done-read-joystick)
    (let* ((joystick-data (read-simulated-joystick))
            (u (first joystick-data))
            (v (second joystick-data))
            (r (third joystick-data))
            (old-u (first present-joystick-command))
            (old-v (second present-joystick-command))
            (old-r (third present-joystick-command)))
      ...)}
(old-r (third present-joystick-command))
(if (or deadlock (> deceleration-factor 30))
 (if (and (<= (* (signum u) (signum old-u)) 0)
 (<= (* (signum v) (signum old-v)) 0)
 (<= (* (signum r) (signum old-r)) 0))
 (progn (setf deadlock nil)
 (setf deceleration-factor 0)
 (setf exit-flag T))
 (display))
 (setf exit-flag T))
 (setf present-joystick-command (list u v r))))
(defun display-cursor ()
  (print "Do you want to random terrain? If so, hit 0 key."
  (if (equal (read) 0)
    (random-terrain)
    (do ((joy-data nil) (x nil) (y nil) (r nil) (fire nil)
      (exit-flag nil))
      (exit-flag (erase-cursor (list cursor-x cursor-y)))
      (zl-user:make-visible)
      (setf joy-data (read-simulated-joystick))
      (setf x (- (second joy-data)) (setf y (first joy-data))
      (setf r (third joy-data)) (setf fire (fourth joy-data))
      (erase-cursor (list cursor-x cursor-y))
      (cond
        (fire (setf exit-flag t))
        ((> x 0) (setf cursor-x (+ cursor-x 1)) (if (> cursor-x 39)
          (setf cursor-x 39)))
        ((< 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 39)
          (setf cursor-y 39)))
        ((< 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)))
      (draw-cursor (list cursor-x cursor-y))
      (draw-obstacles)
      (setf joy-x 0) (setf joy-y 0) (setf joy-r 0))))

(defun draw-terrain(eye-space)
  ; external function: \display.library\move-to-earth, draw-to-earth
  (dotimes (x 40)
    (move-to-earth (list x 0 (aref terrain-height-array x)))
    (draw-to-earth (list x 39 (aref terrain-height-array x))))
  (dotimes (y 40)
    (move-to-earth (list 0 y 0))
    (draw-to-earth (list x y (aref terrain-height-array x))))
(defun 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)))))))

(defun erase-terrain(eye-space)
  ; external function: \display.library\move-to-earth, draw-to-earth
  (tv:sheet-force-access (zl-user:*robot-window*)
    (send zl-user:*robot-window* :clear-window))

(defun init-vehicle-globals ()
  (setf terrain-data (make-array '(39 39) :initial-element 0))
  (setf vehicle-points (make-array 28))
  (setf previous-vehicle-points (make-array 28))
  (setf polygons (make-array 13))
  (setf npolygons nil)
  (setf vertices (make-array 100))
  (setf eye-space nil)
  (setf middle-of-screen nil)
  (read-vehicle-data)) ; read-vehicle-data from disk file

(defun read-simulated-joystick ()
  ; (f 102) (b 98) (r 114) (l 108) (+ 61) (- 45) (origin 150) (fire x)
  (let* ((key-value)
          (delta-x 0.2) (delta-y 0.1) (delta-r 0.01))
    (setf key-value (zl-user:get-keyboard-input))
    (cond ((equal key-value 102) (setf joy-x (+ joy-x delta-x)))
          ((equal key-value 98) (setf joy-x (- joy-x delta-x)))
          ((equal key-value 114) (setf joy-y (+ joy-y delta-y)))
          ((equal key-value 108) (setf joy-y (- joy-y delta-y)))
          ((equal key-value 61) (setf joy-r (+ joy-r delta-r)))
          ((equal key-value 45) (setf joy-r (- joy-r delta-r)))
          (cond ( (>= joy-x 2) (setf joy-x 2))
                 ( (< joy-x -2) (setf joy-x -2))
                 ( (>= joy-y 1) (setf joy-y 1))
                 ( (< joy-y -1) (setf joy-y -1))
                 ( (>= joy-r 0.1) (setf joy-r 0.1))))
(((<= joy-r -0.1) (setf joy-r -0.1)))
(cond ((equal key-value 150) (setf joy-x 0)
(setf joy-y 0) (setf joy-r 0)))
(list joy-x joy-y joy-r (equal key-value 120)))

(defun read-terrain-height ()
  (print "Please input terrain height."
  (setf terrain-height-list (read))
  (let* ((terrain-height-array (make-array 40))
    (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)))
    (terrain-height-list (cdr terrain-height-list))
    (dotimes (i 40)
      (setf zz (+ (* slope (- i x1)) zz))
      (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))))

(defun save-terrain ()
  (draw-obstacles)
  (draw-terrain eye-space)
  (zl-user:save-terrain-to-terrain-buffer))
(defun 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))))

(defun draw-obstacles ()
  ; externals : terrain
  (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 (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))))))

(defun 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)
(defun random-terrain ()
  (let ((a 43411) (b 17) (c 640001) (percent nil) (seed nil) (x nil))
    (print "How much obstacles on the terrain in percentage? ")
    (setf percent (read))
    (print "Seed please")
    (setf seed (read))
    (setf x seed)
    (dotimes (i 39)
      (dotimes (j 39)
        (if (< (/ (setf x (mod (+ (* a x) b) c)) c) (/ percent 100))
          (setf (aref terrain-data i j) 1))))
    (draw-obstacles)
    (zl-user: make-visible))
(defun read-vehicle-data ()
; external 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 "walker:>kwak>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)))
(defun draw-to-earth (a-point)
  (let ((draw-pt (make-displayable
                  middle-of-screen
                  (transform eye-space a-point))))
    (zl-user:draw-to
     (list (truncate (first draw-pt))
           (truncate (second draw-pt)))
     zl-user:*robot-window*))

(defun erase-to-earth (a-point)
  (let ((draw-pt (make-displayable
                   middle-of-screen
                   (transform eye-space a-point))))
    (zl-user:erase-to
     (list (truncate (first draw-pt))
           (truncate (second draw-pt)))
    zl-user:*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))
         (trans nil) (eye nil)
         (radius (first eye-pt)) (alpha (second eye-pt))
         (beta (third eye-pt))
         (center-of-interest (list (/ 39 2) (/ 39 2) 0))
         (transmat 0 0 (- radius))
         (trans eye (matrixmult orient trans))
         (rotatemat 'y (- alpha))
         (trans eye (matrixmult eye rot))
         (rotatemat 'x (- beta))
         (trans 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))))
  (zl-user:move-to
  (list (truncate (first draw-pt))
  (truncate (second draw-pt))))))
(defun display-vehicle (a-H foot-positions)
  ;externals: vehicle-points, previous-vehicle-points, middle-of-screen.eye-space
  (tv:sheet-force-access (zl-user:*robot-window*)
   (send zl-user:*robot-window* :clear-window))
  (body-pento-wrt-earth a-H foot-positions)
  (draw-vehicle vehicle-points))
(defun draw-vehicle ( vehicle-points )
  ; external 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 start))
      (dotimes (j num-vertices)
        (draw-to-earth (aref vehicle-points
                          (aref vertices (+ start j)))))))
(defun transform-body-points (a-H)
 ; externals : body-points, 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)))) 1) 4))

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

(defun vehicle-knee-pos (hipx hipy m l s1 s2 c3 dl dm theta sign1 sign2)
  (let* ((numer (+ (* s1 s1) (- (* s2 s2)) (* dl dl) (* dm dm)))
          (denom (* 2 s1 (sqrt (+ (* dl dl) (* dm dm))))
                 (bet a (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 1)))
                 (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)))))))
    218)
\[\text{(- angle1 angle2))}\]

\[
\text{(defun vehicle-top-pos (hipx hipy m dl theta signl)}
\text{ (let* ((xt hipx)}
\text{ (1-dl (- 1 dl))}
\text{ (sina (sin theta))}
\text{ (cosa (cos theta))}
\text{ (yt (+ (* signl (+ (* m cosa) (* 1-dl sina))) hipy))}
\text{ (zt (- (* m sina) (* 1-dl cosa))))}
\text{ (list xt yt zt))))}
\]
(defun check-event (legs)
  (check-contact legs) ; simulate contact sensor
  (make-contact-state-legs legs) ; feedback variables for synchronize
  (make-ready-state-legs legs) ; between execution and planning
  (make-support-state-legs legs);
)

(defun leg-move (legs)
  (mapcar 'move legs))

(defun leg-state-change (legs)
  (mapcar 'state-transition legs))
(defun check-contact (legs)
    ; this routine checks sensors
    ; Present routine simulate contact sensors.
    ; external function : to-earth-transform
    ; external variable : H1
    (do ((legs legs (cdr legs)))
        ((null legs) nil)
        (let* ((a-leg (car legs))
                (a-leg-pos-wrt-earth (to-earth-transform H1 (leg-pos a-leg)))
                (terrain-height-under-a-leg (find-terrain-height a-leg-pos-wrt-earth)))
          (if (<= (third a-leg-pos-wrt-earth) (+ terrain-height-under-a-leg 0.02))
              (setf (leg-contact-confirm a-leg) T)
              (setf (leg-contact-confirm a-leg) nil))))

(defun find-terrain-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))
            (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 x1))
                    (z2 (aref terrain-height x2))
                    (slope (* z2 z1))
                    (del-x (- x x1)))
              (+ z1 (* slope del-x)))))))
(defun make-contact-state-legs (legs)
  ; external variables : contact-state-legs
  (let ((contacted-legs nil))
    (dolist (a-leg legs)
      (if (equal (leg-state a-leg) 'contact)
        (setf contacted-legs (cons (leg-name a-leg) contacted-legs)))
    (setf contact-state-legs contacted-legs)))

(defun make-ready-state-legs (legs)
  ; external variables : ready-state-legs
  (let ((ready-legs nil))
    (dolist (a-leg legs)
      (if (equal (leg-state a-leg) 'ready)
        (setf ready-legs (cons (leg-name a-leg) ready-legs)))
    (setf ready-state-legs ready-legs)))

(defun make-support-state-legs (legs)
  ; external variables : support-state-legs
  (let ((support-legs nil))
    (dolist (a-leg legs)
      (if (equal (leg-state a-leg) 'support)
        (setf support-legs (cons (leg-name a-leg) support-legs)))
    (setf support-state-legs support-legs)))
(setf (get 'contact 'how-to-move)
  ; externals body-trans-rate1, body-rotate-rate1
  #'(lambda (a-leg)
      (let ((leg-velocity-wrt-body (find-velocity-wrt-body a-leg)))
        (setf (leg-pos a-leg)
              (vectadd (magvect sampling-time leg-velocity-wrt-body)
                        (leg-pos a-leg))))))

(setf (get 'deploy 'how-to-move)
  ; externals : body-trans-rate1
  ; external function :\execute\check_ev\find_ter
  #'(lambda (a-leg)
       (let* ((desired-pos (desired-pos-for-deploy a-leg))
              (dt (- T1 (leg-time a-leg))))
         (setf (leg-pos a-leg)
               (move-del desired-pos (leg-pos a-leg) dt))))

(setf (get 'descent 'how-to-move)
  #'(lambda (a-leg)
       (let ((dt (- planned-contact-time (leg-time a-leg))))
         (if (> (leg-time a-leg) planned-contact-time)
             (setf (leg-pos a-leg)
                   (to-body-transform inv-H1
                                    (leg-desired-pos a-leg))
                   (setf (leg-pos a-leg)
                         (move-del
                          (to-body-transform inv-H1
                                         (leg-desired-pos a-leg))
                          (leg-pos a-leg) dt))))))

(defun desired-pos-for-deploy (a-leg)
  223
; returns desired-pos-wrt-body in deploy state
; external variable : H1, inv-H1
; external function : to-earth-transform, to-body-transform
(let* ((desired-pos-wrt-earth (leg-desired-po3 a-leg))
   (terrain-height (find-terrain-height desired-pos-wrt-earth))
   (desired-pos-height-wrt-earth (+ terrain-height 1.4))
   (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)))

(defun find-velocity-wrt-body (a-leg)
 ; returns foot-velocity-wrt-body
 ; externals : body-trans-rate, body-rotate-rate
 (vectsub '(0 0 0)
          (vectadd body-trans-rate1
                   (crossprod body-rotate-rate1
                              (leg-pos a-leg)))))

(setf (get 'lift 'how-to-move)
      '#(lambda (a-leg)
          (let* ((dt (- T3 (leg-time a-leg)))
                 (desired-pos (lift-pos-desired a-leg))
                 (z (third desired-pos))
                 (ready-pos (leg-ready-position a-leg)))
            (setf (leg-pos a-leg)
                  (move-del desired-pos (leg-pos a-leg) dt))
            (setf (leg-ready-position a-leg)
                  (list (first ready-pos) (second ready-pos) z)))))

(defun lift-pos-desired (a-leg)
 ; returns position-wrt-body which will be at the end of lift state.
 (let* ((leg-pos-wrt-body (leg-pos a-leg))
        (leg-pos-wrt-earth (to-earth-transform H1 leg-pos-wrt-body))
        (desired-height (* 1.4 (find-terrain-height leg-pos-wrt-earth)))
        (to-body-transform inv-H1 (list (first leg-pos-wrt-earth)
                                         (second leg-pos-wrt-earth)
                                         desired-height)))

(defun move (leg-name)
   (apply (get (leg-state leg-name) 'how-to-move)
          (list leg-name)))
(defun move-del (desired-pos present-pos dt)
  (if (< dt 0.05)
      desired-pos
      (let* ((inv-time-diff (/ 1 dt))
             (del (vectsub desired-pos present-pos))
             (velocity (magvect inv-time-diff del))
             (vectadd present-pos (magvect sampling-time velocity))))

(setf (get 'ready 'how-to-move)
      #'(lambda (a-leg)
          (setf (leg-pos a-leg) (leg-ready-position a-leg))))

(setf (get 'return 'how-to-move)
      #'(lambda (a-leg)
          (let ((dt (- T4 (leg-t:me a-leg)))
                (desired-pos (leg-ready-position a-leg)))
            (setf (leg-pos a-leg)
                  (move-del desired-pos (leg-pos a-leg) dt))))

(setf (get 'support 'how-to-move)
:externals :body-trans-rate1, :body-rotate-rate1
      #'(lambda (a-leg)
          (let ((leg-velocity-wrt-body (find-velocity-wrt-body a-leg))
                (leg-pos a-leg)
                (vectadd (magvect sampling-time leg-velocity-wrt-body)
                         (leg-pos a-leg))))

225
(setf (get 'contact 'how-to-change)
  #'(lambda (leg-name)
      (if (> (leg-time leg-name) T2)
          (setf (leg-state leg-name) 'support)
          (if (member-of-recover-command leg-name)
              (progn (setf (leg-state leg-name) 'lift)
                      (setf (leg-contact-confirm leg-name) nil)
                      (setf (leg-time leg-name) zero-time))))))

; Emergency situation: a leg reaches the kinematic limit.
; When leg cycling is in full speed, support state can be skipped.
; Because of two reasons there should be two tests. One test is checking
; timed event, the other test is checking asynchronous event to handle
; above two cases.

(setf (get 'deploy 'how-to-change)
  #'(lambda (leg-name)
      (if (> (leg-time leg-name) T1)
          (progn (setf (leg-state leg-name) 'descent)
                  (setf (leg-time leg-name) zero-time))))))

(setf (get 'descent 'how-to-change)
  #'(lambda (leg-name)
      (if (leg-contact-confirm leg-name)
          (progn (setf (leg-state leg-name) 'contact)
                  (setf (leg-time leg-name) zero-time))))))

(setf (get 'lift 'how-to-change)
  #'(lambda (leg-name)
      (if (> (leg-time leg-name) T3)
          (progn (setf (leg-state leg-name) 'return)
                  (setf (leg-time leg-name) zero-time))))))
(defun member-of-recover-command (a-leg)
  (let ((a-leg-name (leg-name a-leg)))
    (if (member a-leg-name (get 'command 'recover))
        t
      nil)))

(setf (get 'ready 'how-to-change)
      #'(lambda (a-leg)
          (if (get 'command 'deploy) ; if deploy command has leg name
              (if (equal (get 'command 'deploy) (leg-name a-leg))
                  (progn (setf (leg-state a-leg) 'deploy)
                          (setf (leg-time a-leg) zero-time))))))

(setf (get 'return 'how-to-change)
      #'(lambda (leg-name)
          (if (>= (leg-time leg-name) T4)
              (setf (leg-state leg-name) 'ready))))

(defun state-transition (leg-name)
  (apply (get (leg-state leg-name) 'how-to-change)
         (list leg-name)))

(setf (get 'support 'how-to-change)
      #'(lambda (leg-name) ; field recover has a list of leg names.
          (if (member-of-recover-command leg-name)
              (progn (setf (leg-state leg-name) 'lift)
                      (setf (leg-contact-confirm leg-name) nil)
                      (setf (leg-time leg-name) zero-time))))))

227
(defun arc-cos (s)
  (let ((c (sqrt (- 1 (* a a)))))
    (atan2 c a)))

(defun atan2 (y x)
  (if (> (abs x) 0.000001) ; not zero
    (if (> x 0)
        (atan (/ y x))
        (+ (* (atan (/ y x) (* (signum y) PI)))
            (* (signum y) (/ PI 2)))))

(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 (+ 1 i)))
      ((null a-list) i)))

(defun crossprod (vect1 vect2)
  ; generate unit normal 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))))
    (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)
  '(progl (car .queue)
           (setf .queue (cdr .queue))))

(defun dotprod (vect1 vect2) ; No dimension limitation !!!
  (do ((vect1 vect1 (cdr vect1))
       (vect2 vect2 (cdr vect2))
       (sum 0))
      ((null vect1) sum)
    (setf sum (+ sum (* (first vect1) (first vect2)))))

(defmacro enqueue (queue-name element)
  ; externals : 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))))

(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)
  (let ((a (first a-vector))
         (b (second a-vector))
         (c (third a-vector))
         (sqrt (+ (* a a) (* b b) (* c c)))))

229
(defun magvect (const vect)
  ; magvect = const * vect
  (let ((a (first vect))
        (b (second vect))
        (c (third vect)))
    (list (* const a) (* const b) (* const c))))

(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.
    (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 normalize-vector (a-vector)
  (let* ((a (first a-vector))
         (b (second a-vector))
         (c (third a-vector)))
    230
(m (magnitude a-vector)))
(if (< m 0.0000001)
  (list 0 0 0)
  (list (/ a m) (/ b m) (/ c m))))

(defun orthogonalization (mt)
  ; Gram-Schmit orthogonalization process
  (let* ((mx
    (tx (aref m t 0 3)) (ty (aref m t 1 3)) (tz (aref m t 2 3))
    (xl (aref m t 0 0)) (x2 (aref m t 0 1)) (x3 (aref m t 0 2))
    (yi (aref m t 1 0)) (y2 (aref m t 1 1)) (y3 (aref m t 1 2))
    (zl (aref m t 2 0)) (z2 (aref m t 2 1)) (z3 (aref m t 2 2))
    (m 1 (magnitude (list x1 y1 z1)))
    (x1 (/ x1 m))
    (y1 (/ y1 m))
    (z1 (/ z1 m))
    (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.

232
(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 (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 (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 (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 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)
(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 i j))))
    matrix))

(defun unit-crossprod (vectl vect2)
  ; generate unitnormal vector of vectl X vect2
  (let* ((xl (first vectl)) (x2 (first vect2))
         (yl (second vectl)) (y2 (second vect2))
         (zl (third vectl)) (z2 (third vect2))
         (x (- (* yl z2) (* y2 zl)))
         (y (- (* x2 zl) (* xl z2)))
         (z (- (* xl y2) (* x2 yl)))
         (m (sqrt (+ (* x x) (* y y) (* z z))))
         (list (/ x m) (/ y m) (/ z m))))

(defun vectadd (vectl vect2)
  ; vectsub = vectl - vect2
  ; no limit in dimension
  (do ((vectl vectl (cdr vectl))
       (vect2 vect2 (cdr vect2))
       (outvect nil))
      ((null vectl) (reverse outvect))
    (setf outvect (cons (+ (first vectl) (first vect2)) outvect))))

(defun vectsub (vectl vect2)
  ; vectsub = vectl - vect2
  ; no limit in dimension
(do ((vect1 vect1 (cdr vect1))
    (vect2 vect2 (cdr vect2))
    (outvect nil))
  ((null vect1) (reverse outvect))
  (setf outvect (cons (- (first vect1) (first vect2)) outvect))))
(defun calculate-motion ()
  ; generate: estimated-support-plane(-wrt-body),
  ; : all (1,6,10) H, inv-H, body-trans-rate, body-rotate-rate
  ; external variables : present-joystick-command, legs,
  ; deceleration-factor
  (body-control present-joystick-command legs deceleration-factor))

(defun change-state (a-leg target-state)
  ; externals : deploy-flag, supporting-legs
  ; leg-projected-state, leg-projected-time, zero-time
  (if (equal target-state 'planned-contact)
      (setf deploy-flag a-leg)) ; only one leg can be deployed
  ; deploy-flag has leg-name
  (setf (leg-projected-state a-leg) target-state))
  ; deploy-flag is a mean to communicate with function generate-command
  ; recover-flag is set by state-transition-by-event

(defun decrease-deceleration-factor ()
  (setf deceleration-factor (- deceleration-factor 1))
  (if (< deceleration-factor 0)
      (setf deceleration-factor 0)))

(defun delete-leg (leg-name)
  ; external : deleted-leg, supporting-legs
  (setf supporting-legs (remove leg-name supporting-legs))
  (setf deleted-leg leg-name))

(defun emergency-lift ()
  ; If any leg at the limit of kinematic margin
  ; then delete leg & inform globally. This is emergency action.
  ; external variable : deleted-leg
  (if (delete-leg (TKM-limit eligible-to-lift-legs))
    (setf deleted-leg (TKM-limit eligible-to-lift-legs))))
(increase-deceleration-factor))

(defun generate-command ()
  ; external variables : legs, deploy-flag (has name of leg)
  ; external functions : change-state
  ; library functions : enqueue, dequeue

  (setf (get 'command 'deploy) nil) ; this comes after stop-wait
  (setf (get 'command 'recover) nil)
  (if deploy-flag ; Does state is changed to planned-contact?
      (set-deploy-command)) ; Generate deploy command.
  (if recover-flag ; Does state is changed to actual-lift?
      (set-recover-command recover-flag)) ; Generate recover command.
  (setf deploy-flag nil)
  (setf recover-flag nil)) ; restore flags (clear)

(defun increase-deceleration-factor ()
  (setf deceleration-factor (+ deceleration-factor 1)))

(defun lift-exchange()
  ; external variable : supporting-legs, deleted-leg
  ; external function : delete-leg, stability, change-state

  (if deleted-leg (change-state deleted-leg 'planned-lift))

  (if (delete-leg (smallest-TKM-leg eligible-to-lift-legs))
      ; delete leg with smallest TKM from eligible-to-lift-legs
      (if (stability safety-margin)
          ; temporary to permanent change for deleted leg
          (change-state deleted-leg 'planned-lift)
          (try-exchange)))
  (decrease-deceleration-factor))

(defun place()
  ; external function : change-state
  ; from \plan\lift-exc\select-largest-stability-leg
  ; from \plan\search-u\select-footholds
  ; from \plan\lift-exc\restore-deleted-leg

  (if (select-largest-stability-leg available-legs)
      (progn
        (change-state largest-stability-leg 'planned-contact)
        (if deleted-leg (change-state deleted-leg 'planned-lift))
        (decrease-deceleration-factor))
(slow-down-vehicle-or-use-new-footholds))

(defun search-update-TKM ()
  ; external variable : available-legs, supporting-legs
  (select-footholds available-legs)
  (update-TKM supporting-legs))

(defun stability (safety-margin)
  (if (>= (calculate-stability) safety-margin)
      t
    nil)

(defun TKM-limit (legs)
  ; externals : TKM-margin, leg-projected-TKM
  (cond ((null legs) nil)
        ((null (leg-projected-TKM (car legs))) (car legs))
        ((<= (leg-projected-TKM (car legs)) TKM-margin) (car legs))
        (t (TKM-limit (cdr legs)))))

; Value 0.0 can be changed for safety margin.
; This routine detects only one leg out-of-kinematic limit.
; Therefore, this routine may
; be improved in future.

(defun update-state-by-event ()
  ; externals (modified) : available-legs, eligible-to-lift-legs,
  ; supporting-legs
  (mapcar 'state-transition-by-event legs)
  (setf available-legs (find-available-legs legs))
  (setf eligible-to-lift-legs (find-eligible-to-lift-legs legs))
  (setf supporting-legs (find-supporting-legs legs)))
(defun body-control (present-joystick-command legs deceleration-factor)
  ;; external variables: estimated-support-plane wrt earth coordinate
  ;; because all positions in legs are in earth coordinate
  ;; estimated-support-plane-wrt-body,H
  (setf estimated-support-plane (estimate-support-plane legs))
  (setf estimated-support-plane-wrt-body
       (plane-transform estimated-support-plane-wrt-body H))
  (do-body-control estimated-support-plane present-joystick-command
                  deceleration-factor))

(defun do-body-control (estimated-support-plane present-joystick-command
                       deceleration-factor)
  ;; external variables : body-trans-rate1, body-rotate-rate1,H1
  (setf body-trans-rate body-trans-rate1)
  (setf body-rotate-rate body-rotate-rate1)
  (setf H H1)
  (do ((iter 0 (+ iter 1)))
    ((equal iter 10) nil)
    (update-H (body-regulate estimated-support-plane
                        present-joystick-command
                        H deceleration-factor))
    (cond ((equal iter 0)
           (setf body-trans-rate1 body-trans-rate)
           (setf body-rotate-rate1 body-rotate-rate)
           (setf H1 H)
           (setf inv-H1 inv-H))
     ((equal iter 5)
      (setf body-trans-rate6 body-trans-rate)
      (setf body-rotate-rate6 body-rotate-rate)
      (setf H6 H)
      (setf inv-H6 inv-H))
     ((equal iter 9)
      (setf body-trans-rate10 body-trans-rate)
      (setf body-rotate-rate10 body-rotate-rate)
      (setf H10 H)
      (setf inv-H10 inv-H))))

239
(defun body-regulate (plane joystick-command H deceleration-factor)
  ; external variables : body-trans-rate, body-rotate-rate
  (terrain-regulation (get-k-gamma-by-slope plane H))
  (joystick-command-regulation joystick-command deceleration-factor)
  (list body-trans-rate body-rotate-rate))

(defun estimate-support-plane (legs)
  ; constants : alpha 0,1,2,3
  (let* ((footholds-for-estimation (get-footholds legs))
         (constants (get-constants footholds-for-estimation)))
    (make-plane-from-coefficient constants)))

(defun update-H (body-trans-rotate-rate)
  ; external variable (modified) : H, inv-H
  (setf H (orthogonalization
            (get-new-H (get-del-H
                         (get-delta
                          body-trans-rotate-rate))))))
  (setf inv-H (matrixinv H)))
(defun eta-function (eta)
  "external constants from \body_con\body_reg\get_k_ga.lsp : min-eta, max-eta, eta1, eta2"
  (let ((slope (/ (- max-eta min-eta) (- eta2 eta1))))
    (+ min-eta (* slope (- eta eta1)))))

(defun get-k-gamma-by-slope (plane H)
  "external variable(from \body_con\body_reg\get_k_ga.lsp) : min-height, max-height"
  (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) (low-slope plane))
                                  ((< eta eta2) (mid-slope eta plane))
                                  (T (high-slope plane))))
    (list (first k-gamma-desired-height)
       (second k-gamma-desired-height)
       (- (third k-gamma-desired-height) height))))

(defun height-function (eta)
  "external constants from \body_con\body_reg\get_k_ga.lsp : min-height, max-height"
  (let ((slope (/ (- max-height min-height) (- eta2 eta1))))
    (- max-height (* slope (- eta eta1)))))

(defun high-slope (plane)
  "external variable(from \body_con\body_reg\get_k_ga.lsp) : min-height, max-eta"
  (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))

(defun joystick-command-regulation
  (joystick-command deceleration-factor)

; external variables(modified) : body-trans-rate, body-rotate-rate
(if (<= deceleration-factor 0); remove effect of deceleration-factor.
  (setf deceleration-factor 0.5))

(let* ((time-constant 0.5)
        (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)))
        (del-vel-x (/ (- x (first body-trans-rate))
                     time-constant))
        (del-vel-y (/ (- y (second body-trans-rate))
                     time-constant))
        (del-r-vel-z (/ (- r (third body-rotate-rate))
                      time-constant))
        (body-trans-rate-x (+ (* (limiting del-vel-x) sampling-time)
                              (first body-trans-rate)))
        (body-trans-rate-y (+ (* (limiting del-vel-y) sampling-time)
                              (second body-trans-rate)))
        (body-trans-rate-z (+ (* (limiting del-r-vel-z) sampling-time)
                              (third body-trans-rate)))

  (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))
  (setf body-trans-rate (list body-trans-rate-x body-trans-rate-y
                              (third body-trans-rate)))

  (setf body-rotate-rate (list (first body-rotate-rate)
                               (second body-rotate-rate)
                               body-rotate-rate-z))))

242
(defun limiting (del-vel)
  ;  limit acceleration to 3.2174 ft/(sec*sec) or 0.1 G.
  (cond ((> del-vel 3.2174) 3.2174)
         ((< del-vel -3.2174) -3.2174)
         (T del-vel)))

(defun low-slope (plane)
  ;  external variable(from body_con\body_reg\get_k_ga.lsp) :  max-height
  (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)))

(defun mid-slope (eta plane)
  (let* ((plane-unit-normal (first plane))
         (a (first plane-unit-normal))
         (b (second plane-unit-normal))
         (m (sqrt (+ (* a a) (* b b))))
         (desired-eta (eta-function eta))
         (desired-height (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)))
(defun terrain-regulation (k-gamma-delta-height)
  ; external variables (modified) : body-trans-rate, body-rotate-rate
  ; k-gamma-delta-height is ((k.x k.y k.z) gamma delta-height).
  (let* ((gain 5)
      (k (first k-gamma-delta-height))
      (gamma (second k-gamma-delta-height))
      (delta-height (third k-gamma-delta-height))
      (body-rotate-rate-x (* gain (first k) gamma))
      (body-rotate-rate-y (* gain (second k) gamma))
      (body-trans-rate-z (* gain delta-height)))
    (setf body-trans-rate (list (first body-trans-rate) (second body-trans-rate) body-trans-rate-z))
    (setf body-rotate-rate (list body-rotate-rate-x body-rotate-rate-y (third body-rotate-rate))))
(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)
      (magvec (/ 1 number-of-points) sum-vect)
      (print "Error in finding average-point of estimate plane")))))

(defun get-a0 (bar-point al)
  (let* ((x-bar (first bar-point))
         (z-bar (third bar-point))
         (- z-bar (* al x-bar))))

(defun get-al (points bar-point common-denominator)
  ; returns a0 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)))))

246
(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))
    (setf footholds (cons (leg-projected-pos a-leg) 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)))
(defun get-delta (body-trans-rotate-rate)
  ; external constant : sampling-time
  (let* ((body-trans-rate (first body-trans-rotate-rate))
         (body-rotate-rate (second body-trans-rotate-rate))
         (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-rotate-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 (delta-trans-rotate)
  ; external function : \library\ident
  ; external variable : H
  (let* ((H-del (ident)) ; initialize identity matrix
          (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 1 2) (- (first delta-rotate)))
    (setf (aref H-del 2 2) 0)
    (setf (aref H-del 0 2) (second delta-rotate))
    (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 (del-H)
  : external variable : H
  (matrixadd H del-H))
(defun set-deploy-command()  
; only one deploy command is possible for one sampling-time  
; external : command, deploy-flag  
  (setf (get 'command 'deploy)  
    (leg-name deploy-flag)); deploy-flag has the leg to be deployed  
  (setf (leg-desired-pos deploy-flag)  
    (leg-projected-pos deploy-flag)))

(defun set-recover-command (leg-names)  
  (setf (get 'command 'recover) leg-names))
(defun compare-TKM()
  (cond ((null (leg-projected-TKM deleted-leg)) T)
        ((< (leg-projected-TKM deleted-leg) TKM-margin) T)
        ((> (leg-projected-TKM largest-stability-leg) (leg-projected-TKM deleted-leg)) T)
        (nil)))

(defun exchange()
  ; all operations affect globally.
  ; projected state: available-leg, planned-contact, eligible-to-lift,
  ; planned-lift, planned-exchange, actual-lift
  (change-state largest-stability-leg 'planned-contact)
  (change-state deleted-leg 'planned-exchange)
  (setf (leg-exchanged-name deleted-leg)
        (leg-name largest-stability-leg))
  ; exchanged-leg has leg-name which will contact on ground.

(defun find-stability (leg-name)
  ; functions: stability\measure-distance, center-of-gravity
  ; convex-hull-points, supporting-points
  ; external variable: H10, supporting-legs
  ; Because stability function uses supporting-legs, leg-name is
  ; temporally added into new-supporting-legs list.
  (if (not (leg-no-cells-available-flag leg-name)) ; if leg has a foothold
      (let ((new-supporting-legs (cons leg-name supporting-legs)))
        (measure-distance (center-of-gravity H10)
                         (convex-hull-points (supporting-points
                                                  (find-order new-supporting-legs))))
       0.0))

(defun restore-deleted-leg ()
  (setf supporting-legs (cons deleted-leg supporting-legs))
  (setf deleted-leg nil))
(defun select-largest-stability-leg (leg-names)
  (if leg-names ; check whether available-legs are nil.
     (do ((leg-names (cdr leg-names) (cdr leg-names))
          (largest-leg (car leg-names) largest-leg))
        ((null leg-names)
         (if (> (find-stability largest-leg) safety-margin)
             (setf largest-stability-leg largest-leg)
             (setf largest-stability-leg nil)))
        (if (< (find-stability largest-leg)
               (find-stability (car leg-names)))
            (setf largest-leg (car leg-names)))))

(defun smallest-TKM-leg (legs)
  ; select smallest-TKM-leg
  (do ((legs (cdr legs) (cdr legs))
       (smallest-leg (car legs))
       (smallest-tkm nil) (tkm -1000))
      ((null legs) smallest-leg)
    (setf smallest-tkm (if (leg-projected-tkm smallest-leg)
                             (leg-projected-tkm smallest-leg) -1000))
    (setf tkm (if (leg-projected-tkm (car legs))
                 (leg-projected-tkm (car legs)) -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")))

(defun try-exchange ()
  ; external variable : largest-stability-margin-leg
  (if (select-largest-stability-leg available-legs)
      ; inform globally the
      ; largest-stability-margin-leg
      (if (compare-TKM)
          ; compare
          (exchange))
      (restore-deleted-leg)))

; restore deleted-leg into support pattern.

253
(defun all-available-legs ()
  (let* ((legs-in-air (delete-list supporting-legs legs))
         (legs-in-air (remove deleted-leg legs-in-air)))
    (if (equal (delete-list available-legs legs-in-air) nil)
        t
        nil)))

(defun all-cells-deleted (available-legs)
  ; check all-cells are deleted for all-eligible-legs
  (if (not available-legs)
      T ; no more legs available
      (do ((legs available-legs (cdr legs))
           (all-cell-deleted-flag t))
           ((null legs) all-cell-deleted-flag)
      (if (not (leg-no-cells-available-flag (car legs)))
          (setf all-cell-deleted-flag nil)))))

(defun choose-second-max (a-leg)
  ; make-second-list makes a list of (pos-list tkm-list).
  (let* ((second-list (make-second-list a-leg))
          (pos-list (car second-list))
          (tkm-list (cadr second-list))
          (pos-tkm (max-pos-tkm pos-list tkm-list)))
    (if (> (second pos-tkm)
            (+ TKM-margin additional-TKM-margin-in-place))
        (progn (setf (leg-projected-permitted-footholds a-leg) pos-list)
                (setf (leg-projected-tkm-list a-leg) tkm-list)
                (setf (leg-projected-pos a-leg) (first pos-tkm))
                (setf (leg-projected-tkm a-leg) (second pos-tkm))
                nil))
    nil)))
(defun deadlock-recovery (legs)
  (select-footholds legs) ; footholds are modified in try-new-foothold
  (let ((useful-legs (delete-list (no-permitted-cell-legs legs) legs)))
    (cond (useful-legs
      (change-state (car useful-legs) 'planned-contact)
      (increase-deceleration-factor)
      (restore-deleted-leg))
    (T (print "Deadlock")
      (setf deceleration-factor 100))))
)

(defun find-tkm-margin-foothold (a-leg)
  (do ((footholds (leg-projected-permitted-footholds a-leg) (cdr footholds))
       (tkm-list (leg-projected-tkm-list a-leg) (cdr tkm-list))
       (out-footholds nil))
    (null footholds)
    (if (>= (car tkm-list)
      (+ TKM-margin additional-tkm-margin-in-place))
      (setf out-footholds (cons (car footholds) out-footholds))))
)

(defun make-second-list (a-leg)
  (let ((max-pos (leg-projected-pos a-leg))
        (pos-list (leg-projected-permitted-footholds a-leg))
        (tkm-list (leg-projected-tkm-list a-leg)))
    (do* ((pos pos-list (cdr pos-list))
          (tkm tkm-list (cdr tkm-list))
          (out-pos list nil) (out-tkm-list nil))
      (null pos-list)
      (if (not (equal max-pos pos))
        (progn (setf out-pos-list (cons pos out-pos-list))
               (setf out-tkm-list (cons (car tkm-list) out-tkm-list))))
      )))

(defun max-pos-tkm (pos-list tkm-list)
  (if (or (null tkm-list) (null pos-list))
    (list ')' -1000)
    (do ((pos pos-list (cdr pos))
         (tkm tkm-list (cdr tkm-list))
         (t"km-margin additional-tkm-margin-in-place))
      (null pos-list)
      (if (>= (car tkm-list)
        (+ TKM-margin additional-tkm-margin-in-place))
        (setf out-footholds (cons (car footholds) out-footholds))))
      )))
)

(defun max-pos-tkm (pos-list tkm-list)
  (if (or (null tkm-list) (null pos-list))
    (list ')' -1000)
    (do ((pos pos-list (cdr pos))
         (tkm tkm-list (cdr tkm-list))
         (t"km-margin additional-tkm-margin-in-place))
      (null pos-list)
      (if (>= (car tkm-list)
        (+ TKM-margin additional-tkm-margin-in-place))
        (setf out-footholds (cons (car footholds) out-footholds))))
      )))
)
(max 0)
(temp-pos nil)
((null pos) (list temp-pos max))
(if (> (car tkm) max)
  (progn
    (setf max (car tkm))
    (setf temp-pos (car pos))))))

(defun no-permitted-cell-legs (legs)
  ; returns legs with no-permitted-cell
  (do ((legs legs (cdr legs))
       (out-legs nil))
      ((null legs) out-legs)
    (if (leg-no-cells-available-flag (car legs))
        (setf out-legs (cons (car legs) out-legs))))

(defun reduced-space-search (leg-names)
  (dolist (a-leg leg-names)
    (if (not (leg-no-cells-available-flag a-leg))
        (if (null (choose-second-max a-leg)) ; nil means no more cells
            (setf (leg-no-cells-available-flag a-leg) t)))))
  ; As a side effect, this function will set no-cells-available-flag
  ; for each leg
  ; if there is no more available cell for placing a leg.

(defun set-largest-stability-foothold (a-leg)
  (do ((footholds (find-tkm-margin-foothold a-leg) (cdr footholds))
       (max-stability -100) (max-foothold nil))
      ((null footholds)
        (if (> max-stability safety-margin)
            (progn
              (setf (leg-projected-pos a-leg) max-foothold)
              (setf (leg-projected-tkm a-leg)
                    (find-tkm (to-body-transform
                               inv-H10
                               max-foothold)
                               a-leg
                               body-trans-rate10
                               body-rotate-rate10))))
        (setf (leg-projected-pos a-leg) (car footholds))
        (if (> (find-stability a-leg) max-stability)
            (setf max-foothold (car footholds))))))

(defun slow-down-vehicle-or-use-new-footholds ()
  (if (not (all-available-legs))

256
(progn (increase-deceleration-factor)
       (restore-deleted-leg))
(try-new-foothold))

(defun try-new-foothold ()
  ; external functions: \plan\lift_exc\select_l
  ; external variables: largest-stability-leg(modify), available-legs
  (do ((exit-flag nil))
      (exit-flag)
    (reduced-space-search available-legs)
    (cond ((all-cells-deleted available-legs)
               (deadlock-recovery available-legs)
               (setf exit-flag T))
          ((select-largest-stability-leg available-legs)
               (change-state largest-stability-leg 'planned-contact)
               (decrease-deceleration-factor)
               (setf exit-flag T))))
(defun select-footholds (legs)
  ; This routine globally assigns foothold and TKM for available legs.
  ; external variables: H6, inv-H6, inv-H10, body-trans-rate10,
  ; body-rotate-rate10
  (do ((legs legs (cdr legs)))
      ((null legs) 'done-select-footholds)
    (find-foothold (car legs) H6 inv-H6 inv-H10 body-trans-rate10
                   body-rotate-rate10)))

(defun update-TKM (legs)
  ; externals: inv-H10, body-trans-rate10, body-rotate-rate10
  (do* ((legs legs (cdr legs))
        (a-leg nil))
    ((null legs) 'done-update-tkm)
    (setf a-leg (car legs))
    (setf (leg-projected-tkm a-leg)
          (find-TKM (to-body-transform
                     inv-H10 (leg-projected-pos a-leg))
                    a-leg
data-trans-rate10 body-rotate-rate10))))
(defun find-foothold (a-leg H6 inv-H6 H10 inv-H10 body-trans-rate10 body-rotate-rate10)
  ; all points are wpt body coordinate system.
  (let* ((four-points (four-points-on-support-plane (leg-four-lines a-leg)))
         (possible-footholds (get-possible-footholds (estimate-footholds four-points a-leg) H6 inv-H6))
         (max-foothold-&-TKM (get-foothold-with-max-TKM possible-footholds a-leg H6 inv-H6 body-trans-rate10 body-rotate-rate10))
         (assign-foothold max-foothold-&-TKM a-leg H6))
)
(defun assign-foothold (foothold-&-TKM a-leg H)
  ; assigns foothold & TKM to a-leg.
  ; external variables : projected-pos, projected-TKM
  ; external function : \find_foo\get_poss\to_earth
  (setf (leg-projected-TKM a-leg) (second foothold-&-TKM))
  (if (first foothold-&-TKM)
      (setf (leg-projected-pos a-leg) (to-earth-transform H (first foothold-&-TKM))))

(defun estimate-footholds (four-points-wrt-body a-leg)
  ; returns estimate-footholds-wrt-body
  ; external variables : leg-sixteen-footholds (two dimensional points)
  (do* ((footholds (leg-sixteen-footholds a-leg) (cdr footholds)) (out-footholds nil) (a-foothold nil))
      ((null footholds) (get-points-on-support-plane out-footholds))
      (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))))

(defun four-points-on-support-plane (four-lines)
  ; returns four points which are intersected by four-lines on
  ; estimated-support-plane-wrt-body
  ; external variables : estimated-support-plane-wrt-body
  ; external function : \library\plane_in
  (do* ((lines four-lines (cdr lines)) (points nil))
      ((null lines) points)
      (setf points (cons (plane-intersection (car lines) estimated-support-plane-wrt-body) points))))
(defun get-foothold-with-max-TKM (possible-footholds a-leg H inv-H body-trans-rate body-rotate-rate)
  ; returns max-foothold-&-TKM
  ; real-footholds is really possible footholds
  (do ((footholds possible-footholds (cdr footholds))
      (foothold nil) (a-foothold nil) (TKM-list nil) (a-TKM nil)
      (real-footholds nil) (max-TKM -100.0))
    ((null footholds) (side-effect-of-get-foothold-with-max-TKM
      real-footholds max-TKM TKM-list a-leg H)
     (list foothold max-TKM))
    (setf a-foothold (car footholds))
    (setf a-TKM (find-TKM a-foothold a-leg 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 foothold a-foothold)))))))

(defun get-possible-footholds (estimated-footholds H inv-H)
  ; returns possible-footholds wrt body
  (to-body-transform inv-H
    (find-possible-footholds
     (to-earth-transform H estimated-footholds))))
(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)
  ; returns intersection points with support plane in z-body direction.
  ; external variables : estimated-support-plane-wrt-body
  ; external function : \library\plane_in
  (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)))))
        (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))))
(defun side-effect-of-get-foothold-with-max-TKM
  (possible-footholds max-TKM TKM-list a-leg H)
  
  ; external variables: no-cells-available-flag, projected-TKM-list, 
  ; projected-permitted-footholds
  ; external functions: \find_foo\get_pos\to_earth
  (setf (leg-no-cells-available-flag a-leg) (< max-TKM 0.0))
  (setf (leg-projected-TKM-list a-leg) TKM-list)
  (setf (leg-projected-permitted-footholds a-leg)
        (reverse (to-earth-transform H possible-footholds))))
(defun find-possible-footholds (estimated-footholds-wrt-earth)
  ; returns possible-footholds-wrt-earth
  ; external variable : terrain
  (do* ((footholds estimated-footholds-wrt-earth (cdr footholds))
        (a-foothold nil) (i-x nil) (i-y nil) (out-footholds nil))
        ((null footholds) (unique-footholds-only out-footholds))
    (setf a-foothold (car footholds))
    (setf i-x (floor (first a-foothold))) ; get terrain x-index
    (setf i-y (floor (second a-foothold))) ; get terrain y-index
    (if (in-side-of-whole-terrain i-x i-y)
      (if (equal (aref terrain i-x i-y) 0) ; permitted cell
       (setf out-footholds
         (cons (terrain-point i-x i-y) out-footholds)))))))

(defun get-terrain-height (i-x)
  (aref terrain-height i-x))

(defun in-side-of-whole-terrain (i-i-y)
  ; externals : terrain
  (let ((dimension-terrain (array-dimensions terrain)))
    (cond ((< i-x 0) nil)
          ((< i-y 0) nil)
          ((> i-x (- (first dimension-terrain) 1)) nil)
          ((> i-y (- (second dimension-terrain) 1)) nil)
          (t))))

(defun terrain-point (i-i-y)
  (list (+ i-x 0.5) (+ i-y 0.5) (get-terrain-height i-x)))

(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) 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) out-points)
        (setf out-points (cons (transform H (car points)) out-points)))
    (transform H points-wrt-body))) ; single point case

(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)))))
(defun find-tkm (a-foothold a-leg body-trans-rate body-rotate-rate)
  ; a-foothold is based on body coordinate
  ; returns tkm
  ; external variables : leg-working-volume
  ; external functions : library\transform
  (let* ( (leg-vel-rpt-body (get-leg-velocity a-foothold body-trans-rate body-rotate-rate))
    (working-volume (leg-working-volume a-leg)))
    (get-tkm a-foothold leg-vel-rpt-body working-volume)))

(defun get-distance (planes velocity leg-position)
  ; external 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)
  ; external function : magnitude
  ; outside w.v returns nil. If speed is 0, then returns 1000.0.
  (if (in-side-volume leg-pos-rpt-body working-volume)
(let ((speed (magnitude velocity)))
  (if (= speed 0)
      1000.0
      (/ (get-distance working-volume velocity leg-pos-rpt-body) speed)))

(defun in-side-volume (position planes)
  (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))
(defun calculate-stability ()
  ;; externals : H1O
  (if (>= (counting supporting-legs) 3)
    ;; measure-distance (center-of-gravity H1O)
    (convex-hull-points
     ;; (supporting-points
     (find-order supporting-legs)))
    0.0))

(defun center-of-gravity (H)
  ;; center-of-body is represented wrt earth coordinate.
  (let ((x (aref H 0 3))
    (y (aref H 1 3))
    (z (aref H 2 3)))
    (list x y z)))

(defun convex-hull-points (points)
  ;; point is a list(x y z). For a time being, only x,y are used.
  ;; external function : delete-list
  (if (> (counting points) 3)
    (let* ((boundary-points (out-side points))
      (remaind (delete-list boundary-points points)))
      (cond (remaind (convex-hull-points boundary-points))
        (T boundary-points)))
    points)) ; minimum points (3) are reached.

(defun distance (point1 point2)
  (let* ((del-x (- (car point1) (car point2)))
    (del-y (- (cadr point1) (cadr point2))))
    (sqrt (+ (* del-x del-x) (* del-y del-y)))))
(defun find-order (legs)
  ; find ordered leg-names for calculating convex-hull-points
  ; externals : convex-hull-order (This has only ready position for leg
  ; names)
  (let ((ordered-legs nil))
    (dolist (a-leg-name convex-hull-order)
      (dolist (a-leg legs)
        (if (equal a-leg-name (leg-name a-leg))
            (enqueue ordered-legs a-leg))))
    ordered-legs))

(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))))

270
(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)
              -100.0))) ; 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 out-side (points)
  ; this function does not change the order of points except deletion.
  (do* ((first-points points (cdr first-points))
        (second-points (reverse (cons (car points)
                                      (reverse (cdr points)))))
        (third-points (reverse (cons (car second-points)
                                     (reverse (cdr second-points)))))
        (out-points nil out-points))
    ((null first-points)
      (let ((return-points nil))
        (dolist (a-point points)
          (if (member a-point out-points)
              (enqueue return-points a-point))
          return-points)
        (if (test-out-side (car first-points) (car second-points)
                            (car third-points))
          271)
(setf out-points (cons (car second-points) out-points))))

(defun point-distance (center-point first-point second-point)
  (let* ((slopel (find-slope first-point second-point))
         (slope2 (right-angle slopel)))
    (distance center-point
     (intersection-point (line slopel first-point)
                        (line slope2 center-point))))

(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) ; infiniti dummy number 10000.0
        (distance nil))
    ((null first-points) min-distance)
    (setf distance (point-distance center-point
                                    (car first-points) (car second-points)))
    (if (< distance min-distance)
        (setf min-distance distance))))

(defun supporting-points (leg-names) ; projected supporting-points
 ; externals: leg-projected-pos
  (let ((leg-names-in-order (find-order leg-names)))
    (do* ((leg-names-in-order leg-names-in-order (cdr leg-names-in-order))
          (a-leg (car leg-names-in-order) (car leg-names-in-order))
          (points nil))
      ((null leg-names-in-order) points)
      (enqueue points (leg-projected-pos a-leg))))

(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))))
    272)
(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)))
(defun calculate-present-stability (a-leg)
  ; check present stability without a-leg.
  ; external functions : \plan\stabilit\find_ord, convex_h, 
  ;                      center_of, measure_
  (let* ((supporting-legs (find-present-supporting-legs legs))
          (net-supporting-legs (remove a-leg supporting-legs)))
    (if (>= (counting net-supporting-legs) 3)
        (measure-distance (center-of-gravity H1)
                         (convex-hull-points
                          (present-supporting-points
                           (find-order net-supporting-legs)))
                         0.0))
    0.0))

(defun find-available-legs (legs)
  ; external variable : available-legs
  (do ((legs legs (cdr legs))
       (out-legs nil))
      (null legs)
    (if (equal (leg-projected-state (car legs)) 'available-leg)
        (setf out-legs (cons (car legs) out-legs)))))

(defun find-eligible-to-lift-legs (legs)
  ; returns eligible-to-lift-legs
  (do ((legs legs (cdr legs))
       (out-legs nil))
      (null legs)
    (if (equal (leg-projected-state (car legs)) 'eligible-to-lift)
        (setf out-legs (cons (car legs) out-legs))))

(defun find-present-supporting-legs (legs)
  ; returns present-supporting-legs
  (do ((legs legs (cdr legs))
       (out-legs nil))
      (null legs)
    (if (equal (leg-projected-state (car legs)) 'present-supporting)
(a-leg nil)
((null legs) out-legs)
(setf a-leg (car legs))
(if (or (equal 'contact (leg-state a-leg))
        (equal 'support (leg-state a-leg)))
    (setf out-legs (cons a-leg out-legs))))

(defun find-supporting-legs (legs)
  ; returns a list of supporting-legs
  (do ((legs legs (cdr legs))
       (leg-names nil leg-names))
      ((null (car legs)) leg-names)
    (let ((state-of-leg (leg-projected-state (car legs))))
      (if (or (equal state-of-leg 'planned-contact)
              (equal state-of-leg 'eligible-to-lift))
          (setf leg-names (cons (car legs) leg-names))))))

(defun interlock-confirm (a-leg)
  ; external variable : contact-state-legs
  (if (member (leg-exchanged-name a-leg) contact-state-legs)
    T
    nil))

(defun present-supporting-points (leg-names) ; present supporting-points
  ; externals: leg-pos-x,y,z, H1
  (let ((leg-names-in-order (find-order leg-names)))
    (do* ((leg-names-in-order leg-names-in-order
           (cdr leg-names-in-order))
          (a-leg (car leg-names-in-order) (car leg-names-in-order))
          (points nil points))
        ((null leg-names-in-order) points)
      (enqueue points (to-earth-transform H1 (leg-pos a-leg))))))

(defun stable-without (a-leg)
  (if (>= (calculate-present-stability a-leg) safety-margin)
    t
    nil))

(defun state-transition-by-event (a-leg)
  ; external variables : ready-state-legs, contact-state-legs,
support-state-legs
; these variables from execute block or physical leg for synchronization
; and it also provides a way to feedback from outside to planning block.
(let* ((leg-planning-state (leg-projected-state a-leg))
   (leg-name (leg-name a-leg)))
  (cond ((and (equal leg-planning-state 'actual-lift)
              (member leg-name ready-state-legs))
         (setf (leg-projected-state a-leg) 'available-lsg))
       ((and (equal leg-planning-state 'planned-contact)
              (member leg-name contact-state-legs))
         (setf (leg-projected-state a-leg) 'eligible-to-lift))
       ((and (equal leg-planning-state 'planned-lift)
              (member leg-name support-state-legs)
              (stable-without a-leg))
         (setf (leg-projected-state a-leg) 'actual-lift)
         (setf (leg-exchanged-name a-leg) nil)
         (setf recover-flag (cons (leg-name a-leg) recover-flag)))
       ((and (equal leg-planning-state 'planned-exchange)
              (member leg-name support-state-legs)
              (interlock-confirm a-leg))
         (setf (leg-projected-state a-leg) 'actual-lift)
         (setf (leg-exchanged-name a-leg) nil)
         (setf recover-flag (cons (leg-name a-leg) recover-flag)))))

; recover-flag has leg-names such as leg1, leg2
; recover-flag is a mean to communicate with function generate-command.
B.2 Graphics Primitives
(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)

(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 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))

278
(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
    :edges-from :mouse
    :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)))
B.3 Data File for Simulation Vehicle Model
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)
REFERENCES


283


285


