

Rapid Generation of Motion Plans for Modular Robotic Systems

by
Jeffrey R. Cole

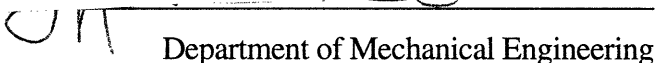
B.S., Mechanical Engineering
Massachusetts Institute of Technology, 1993

Submitted to the Department of Mechanical Engineering in Partial
Fulfillment of the Requirements for the Degree of

Master of Science
at the
Massachusetts Institute of Technology
May 12, 1995

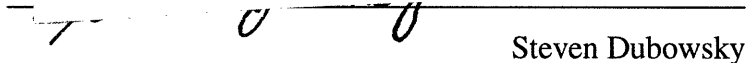
©1995 Massachusetts Institute of Technology. All rights reserved.

Signature of Author


Department of Mechanical Engineering

May 12, 1995

Certified by



Steven Dubowsky
Thesis Supervisor

Accepted by



Ain A. Sonin
Chairman, Department Graduate Committee

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

AUG 31 1995

LIBRARIES

Barker Eng

Rapid Generation of Motion Plans for Modular Robotic Systems

by

Jeffrey R. Cole

Submitted to the Department of Mechanical Engineering on May 12, 1995 in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

Robots are needed to perform important tasks in field environments. A limitation to the practical use of robot systems is their high cost and long development times. It would be desirable to rapidly design and fabricate the hardware and software for these systems, on a time scale of days or weeks instead of years.

An integral part of this system is a rapid generator of the motion plans that the robot will execute in order to perform the task. The goal of this research is to develop a methodology for generating these motion plans rapidly, and with as little user intervention as possible.

This thesis describes one approach to rapid generation of motion plans for robotic systems operating in field environments. The approach assembles software modules which control robot actions into executable "scripts". For each application, an inventory of modules is provided based on task characteristics. A script's performance is evaluated using off-line simulation of the application robot and environment. Successful scripts for a specific task are found using the genetic algorithm search technique.

The thesis discusses the implementation of this approach. The approach is applied to an example restoration task aboard the USS Constitution.

The results of this application demonstrate that the scripts can be trained to succeed over the range of characteristic field environment variability. Also, one action module inventory may generate successful scripts for various robot configurations. Furthermore, successful scripts can be generated more quickly by including action modules which solve task-specific problems.

Supervisor: Dr. Steven Dubowsky

Title: Professor of Mechanical Engineering

Table of Contents

Acknowledgments	6
Chapter 1 Introduction	7
<i>1.1 Motivation</i>	7
<i>1.2 Approach to rapid plan generation</i>	9
<i>1.3 Related literature</i>	11
Chapter 2 Overview: the Rapid-deployment Modular Robotic System	13
<i>2.1 Rapid deployment system: the modular approach</i>	13
<i>2.2 The rapid designer</i>	14
<i>2.3 The rapid plan generator</i>	17
2.3.1 Creating the inventory of action modules	17
2.3.2 Optimizing the script	19
<i>2.4 The control system generator</i>	20
Chapter 3 Rapid motion plan generation using genetic programming	21
<i>3.1 Why use genetic programming?</i>	21
<i>3.2 The genetic algorithm</i>	22
<i>3.3 The genes for robot motion plans: “action modules”</i>	25
3.3.1 The content of an action module	27
3.3.2 Creating the inventory of action modules for a specific robot application	30
<i>3.4 Evaluating a script</i>	32
<i>3.5 Creating offspring scripts</i>	37
<i>3.6 Picking good training problems to improve robustness</i>	40
Chapter 4 A sample application--restoring the USS Constitution	42
<i>4.1 The ballast decking inspection task</i>	42
<i>4.2 Modeling the Constitution application for rapid plan generation</i>	48
4.2.1 Environmental model	49
4.2.3 Robot model	49
4.2.2 The task description	52
<i>4.3 Creating the rapid plan generator for the Constitution task</i>	54
4.3.1 Creating an inventory of action modules	54
4.3.2 Simulating a script’s performance	58
4.3.3 Defining the fitness function	58
4.3.4 Acknowledgment: authorship of the genetic algorithm software	60
Chapter 5 Results from the Constitution application	62
<i>5.1 A typical solution</i>	62

<i>5.2 Improving robustness through training on worst-case problems</i>	66
<i>5.3 Improving success by including useful “higher-level” modules in the inventory</i>	72
<i>5.4 Applicability of an inventory to varying robot configurations</i>	75
Chapter 6 Conclusions	76
<i>6.1 Summary of Results</i>	76
<i>6.2 Recommendations for future research</i>	76
References	78

Table of Figures

Figure 1 Multi-limbed robots for field environment applications	8
Figure 2 The goal of the rapid deployment robotic system.	8
Figure 3 Assembling action modules into an executable script	10
Figure 4 The rapid deployment system design cycle	15
Figure 5 Some example modular design components and assemblies (courtesy [28])	16
Figure 6 An example of a modular robot	17
Figure 7 Examples of action modules and a script	18
Figure 8 The genetic algorithm cycle	24
Figure 9 An example optimization problem using genetic algorithms	26
Figure 10 Action modules as genes	27
Figure 11 Partial credit in the fitness function	36
Figure 12 Performance cases which require additional fitness function criteria	37
Figure 13 The mechanism of crossover	38
Figure 14 The mechanism of mutation	39
Figure 15 The USS Constitution (courtesy [35])	43
Figure 16 The ballast decking inspection area aboard the Constitution	45
Figure 17 Sensors for the ballast decking inspection task	46
Figure 18 Potential robot configurations for the ballast decking inspection task	48
Figure 19 Model of the Ballast Inspection Area	49
Figure 20 Models of two robots used in the example problem	50
Figure 21 Parameters which describe a robot's position and orientation	51
Figure 22 Static Stability Constraints	53
Figure 23 Environmental Interference Constraints	54
Figure 24 Action modules for controlling body movement	56
Figure 25 Action module for controlling limb movement	57
Figure 26 The simulation cycle for evaluating script performance	59
Figure 27 Robot success as defined by the fitness function.	60
Figure 28 An example script and illustration of the robot executing the script	63
Figure 29 Illustration of robot walking using script	65
Figure 30 Example of a script causing robot failure	66
Figure 31 Ballast area parameters' worst case variation	68
Figure 32 Script performance when applied to cases with environmental variation	69
Figure 33 The environments used to evaluate script robustness	71
Figure 34 Improved success by training on the worst-case problem	72
Figure 35 Example of robot "getting stuck"	73
Figure 36 "Body level" module introduced to prevent getting stuck	73
Figure 37 Reducing cycle time by including Body Level action module	74
Figure 38 Success for different robot configurations using same action module inventory	75

Acknowledgments

This work would not be possible without the guidance and patience of Dr. D. My thanks for the opportunity.

Thanks to the family and friends for the support during the times it all appeared to come to a grinding halt.

Thanks to the many friends in the research group who assisted in technical matters. Special thanks goes to Craig Sunada. Most of all, thanks to Nathaniel Rutman, taught me very much about engineering and more important things.

Thanks to the guys working to restore the USS Constitution, for giving our group a fascinating testbed for our research. Special thanks to Patrick Otton and Charles Deans for the personal attention; the guided tours were unforgettable.

This research was carried out with the partial support of NASA grants #NAG1801 and #NAG11637 and NSF grant #9320138IRI.

Chapter 1 Introduction

This thesis describes an approach to rapid generation of motion plans for modular robotic systems used in unstructured field environments. The approach assembles software modules which control robot actions into executable “scripts”. Optimal scripts for a specified task are found using the genetic algorithm search technique. The thesis discusses the implementation of this method. This method is applied to a restoration task aboard the USS Constitution. The results of this application demonstrate that the scripts can be trained to succeed on the range of environmental variation characteristic of unstructured environments.

This chapter describes the motivation for creating a rapid motion plan generator, the approach taken, and research being done in this field.

1.1 Motivation

(This text for section 1.1 is compiled from other work done by this research group [28,33,34].)

Robots are needed for important missions in field environments [1, 2]. These systems could perform such important tasks as maintenance and disaster mitigation in nuclear power facilities, cleanup of toxic and hazardous waste sites and chemical accident cleanup, terrorist bomb disposal, infrastructure inspection, and commercial tanker hull maintenance [3, 4, 5, 6]. For many of these missions, robotic systems could remove humans from dangerous tasks or enter locations that are not readily accessible. In some applications, such as the inspection of the undersides of highway and rail bridges, robotic systems could also be very cost effective. The concept of the field environment robotic system is shown in figure 1.

A SAMPLE TASK FOR FIELD ROBOT APPLICATIONS
 The robot must repair the field data collection device.

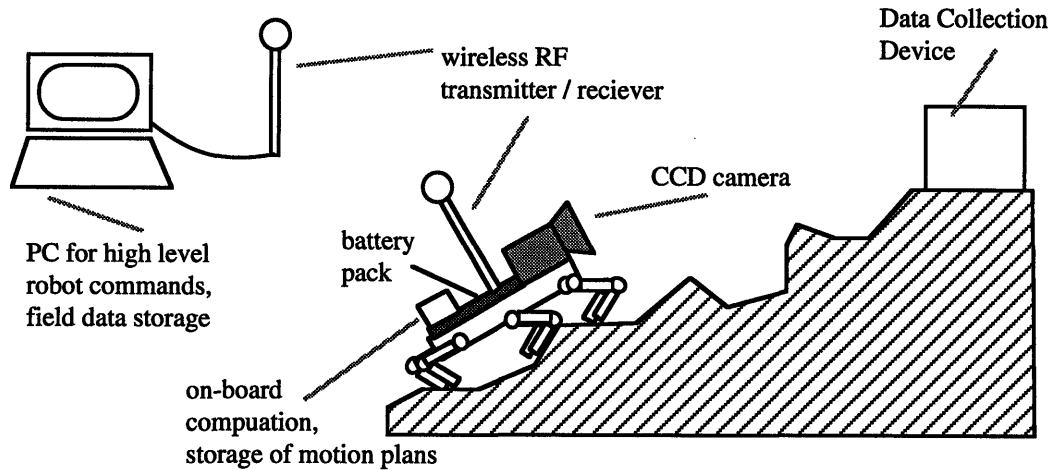


Figure 1 Multi-limbed robots for field environment applications

A major limitation to the practical use of such systems is that their cost using current approaches would be prohibitive for many applications. This is largely because these systems will not generally be mass produced; each system would often need to be designed for a specific mission or task. Not only will their costs be very high but the systems development time for a given mission could take years, when deployment in weeks or months may be required. For these reasons it would be useful to have systems that can be rapidly designed and fabricated. A flowchart illustrating the rapid deployment robotic system goal is illustrated in figure 2.

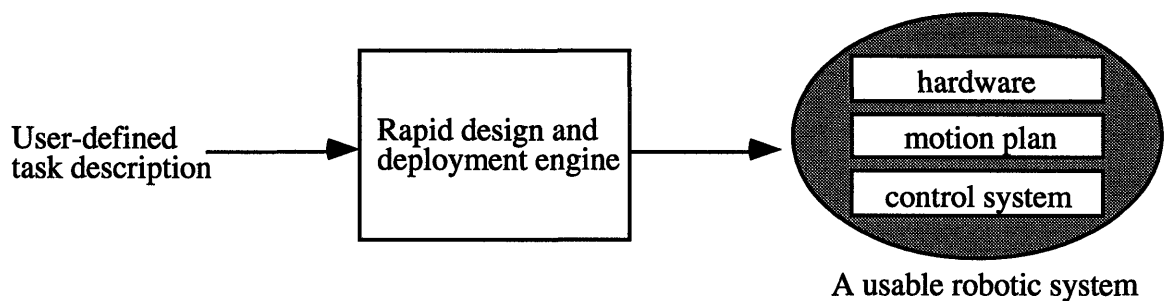


Figure 2 The goal of the rapid deployment robotic system.

Three key components to the robotic system must be generated: the robot design, the motion plan, and the control system. The design specifies what hardware the robot is comprised of, and the manner in which it is assembled. The motion plan dictates the actuator commands that the robot executes to perform the task. The control system consists

of the generalized control variables, control gains, and control algorithm necessary to control the commanded actuator positions and torques.

One reason why conventional robotic systems require such long development time is that the system components are largely developed from scratch for each new application. Significant time is spent designing and fabricating the custom hardware and software required. An approach to rapid system deployment would be to assemble each of these components from an inventory of available modules. No work would need to be done in creating custom parts. The design process for each system component is reduced to finding the optimal module assembly. This process can be largely automated through search techniques.

1.2 Approach to rapid plan generation

An integral part of the rapid deployment system is the rapid plan generator. The approach to plan generation is to assemble a motion plan from an inventory of “action modules” (see figure 3). Each action module contains one or more commands of actuator position or force. The sequence of action modules in the motion plan defines the order in which the robot executes them. The complete sequence of action modules is termed here as a “script.”

It should be noted that the plans generated with this technique dictate not only robot motion, but force application as well. However, the plans will be referred to as “motion plans” throughout this thesis.

The inventory of available action modules is chosen to reflect the needs of the specific application task, robot, and environment. This may be done through the use of experience-based rules.

The advantage to this approach is that no custom software must be written for each application. Instead, a motion plan is created by finding a successful assembly of action modules. This procedure is largely automated through the use of a genetic algorithm-based search technique.

The search process may be terminated when a script demonstrates a level of task success deemed sufficient by the user. This solution may or may not represent the “optimal” script for the task. In fact, there is no formal proof that a genetic algorithm converges towards an

optimal solution [7]. However, for the purposes of this thesis, the rapid plan generator system will be said to search for “optimal” scripts for the task.

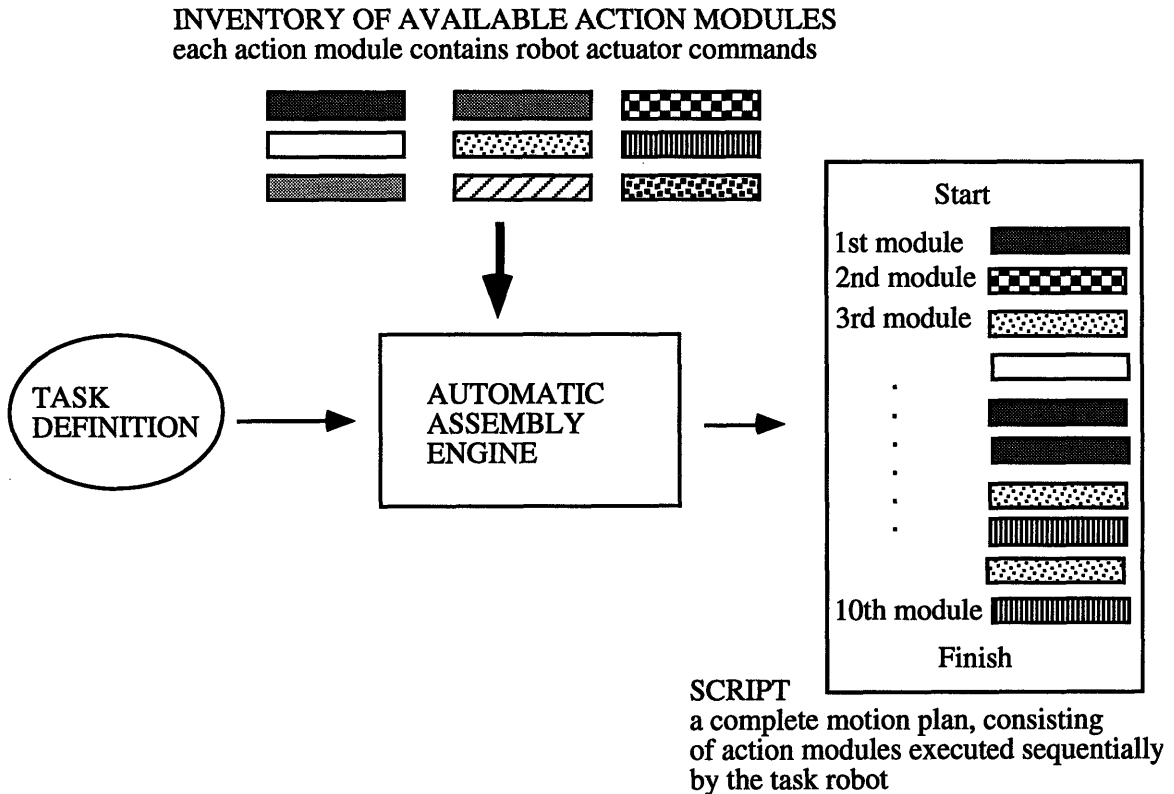


Figure 3 Assembling action modules into an executable script

Field environments present a challenge to motion plan generation in that the nature of the environment may have changed between the time that a robotic system is designed and deployed. For example, a robot cleaning a toxic waste site might encounter additional spillage which did not previously exist. An infrastructure inspection robot investigating the underside of a bridge may find that part of the bridge has deteriorated away. A motion plan generated for a field environment application must accommodate any environmental change that may have occurred by the time the robotic system is deployed. Because of this environmental variability, these field environments are often called “unstructured” environments.

A potential concern with this approach described here is that because the scripts are generated beforehand, they may not succeed when deployed in the actual environment. This thesis presents an original approach to training the scripts to succeed over the range of

predicted environmental variation. The approach is to train the scripts to succeed for the case of maximum environmental variation.

1.3 Related literature

Optimizing robot motion plans requires a search technique which works in highly-dimensional, discontinuous, and non-convex search spaces. Recently, genetic algorithms were developed to provide robust search in such spaces.

The genetic algorithm exploits the biological principles of genetics and evolution to iteratively improve the solutions. The non-analytical nature of the algorithm does not use knowledge of the search space's derivatives, continuity, or modality. The computational simplicity of the algorithm allows a thorough coverage of the search space compared to other search techniques [7].

A great deal of work is being done in using genetic algorithms for solving numerical optimization problems of all types [8,9,10,11]. Example applications include image compression, data filter optimization, curve-fitting, and classification problems. Some work has been done in applying genetic algorithms toward engineering applications [12, 13, 14], including mass/spring/damper system parameter identification [15] and truss optimization [16].

A natural application of genetic algorithms was for the optimization of computer code, which presents a complex search space. This application is known as genetic programming [17,18, 19, 20,21]. Some early applications include the optimization of sorting algorithms, design of control systems, and creation of neural nets.

As a robot motion plan is essentially a computer program, genetic programming has been applied to robot plan optimization. Subsumptive plans [22,23,24] have been generated using genetic programming [25] for wheeled mobile robots.

Some genetic programming work has been done in optimizing robot plans for more complex-model, multi-limbed robots [26]. In this work, a very simple environmental model was used which featured simple geometry and was assumed to be completely known.

Other genetic programming work has focused on developing robot plans which tolerate sensor noise [27]. These plans were not developed to tolerate environment variability

between the training and application problems. In this work also, the robot achieved locomotion via wheels.

Although some robot motion plans have been generated through the use of genetic algorithms, these applications have all been for very structured environments. It has not been proven that the technique can be applied to the class of unstructured environments which features dimensional variability.

Chapter 2 Overview: the Rapid-deployment Modular Robotic System

This chapter describes the rapid-deployment robotic system. The system generates a robot configuration, motion plan, and control system for a specific task on the order of days or weeks. The approach to rapid generation is to design the hardware configuration from a finite inventory of available hardware components, and to design the motion plan from a finite inventory of available actuator command “action modules”.

2.1 Rapid deployment system: the modular approach

A major limitation to the practical use of robotic systems for unstructured environment applications is their prohibitive cost and development times. A solution to this problem is a rapid-deployment system, which designs the robot configuration, motion plans, and control system for a specific task.

Three main functional components are required for such a rapid-deployment system .The first is an automated designer which designs the robot configuration. The configuration consists of the set of all hardware and the manner in which it is assembled. The configuration must adhere to task constraints such as size limitations, required payload, and power consumption limits.

The second component of this system, the motion plan generator, constructs a set of actuator commands which the robot will execute in order to perform the task. The motion plan must also adhere to the task constraints.

The third component, a control system designer, creates a suitable control system for the robot actuators based upon the generated robot configuration and motion plans. The control system consists of an appropriate set of generalized control variables, control gains, and control algorithm.

An approach to rapid system deployment would be to assemble each of these components from an inventory of available modules. No work would need to be done in creating custom parts. The design process for each system component is reduced to finding the optimal module assembly. This process can be largely automated through search techniques.

The rapid deployment system iteratively constructs and evaluates a robot system for the task. The robot design cycle is shown in figure 4. After each cycle, the user may simulate the robotic system's performance to evaluate its success for the actual application task. If the user decides that the robotic system demonstrates sufficient capability, he may terminate the design process and deploy the system. If the user wishes to improve the system design, he may run additional cycles.

Note that additional cycles may or may not yield improved solutions. Currently, there is no formal proof that the system converges towards an optimal solution.

The next section describes the approach for the rapid designer and rapid motion plan generator.

2.2 The rapid designer

The approach taken by Rutman [28] is to assemble robots from a finite inventory of hardware modules. These modules include actuators, end-effectors, motors, power supplies, sensors, and other key items as shown in figure 5. The design process then becomes a matter of searching the possible robot assemblies to find the best configuration for a specified task.

The advantage of this approach is that a search can be automated and executed by software. The total collection of possible assemblies can be represented by a bounded design space. Any optimization technique which works in discrete, discontinuous spaces may be employed to find the best module assembly in the design space.

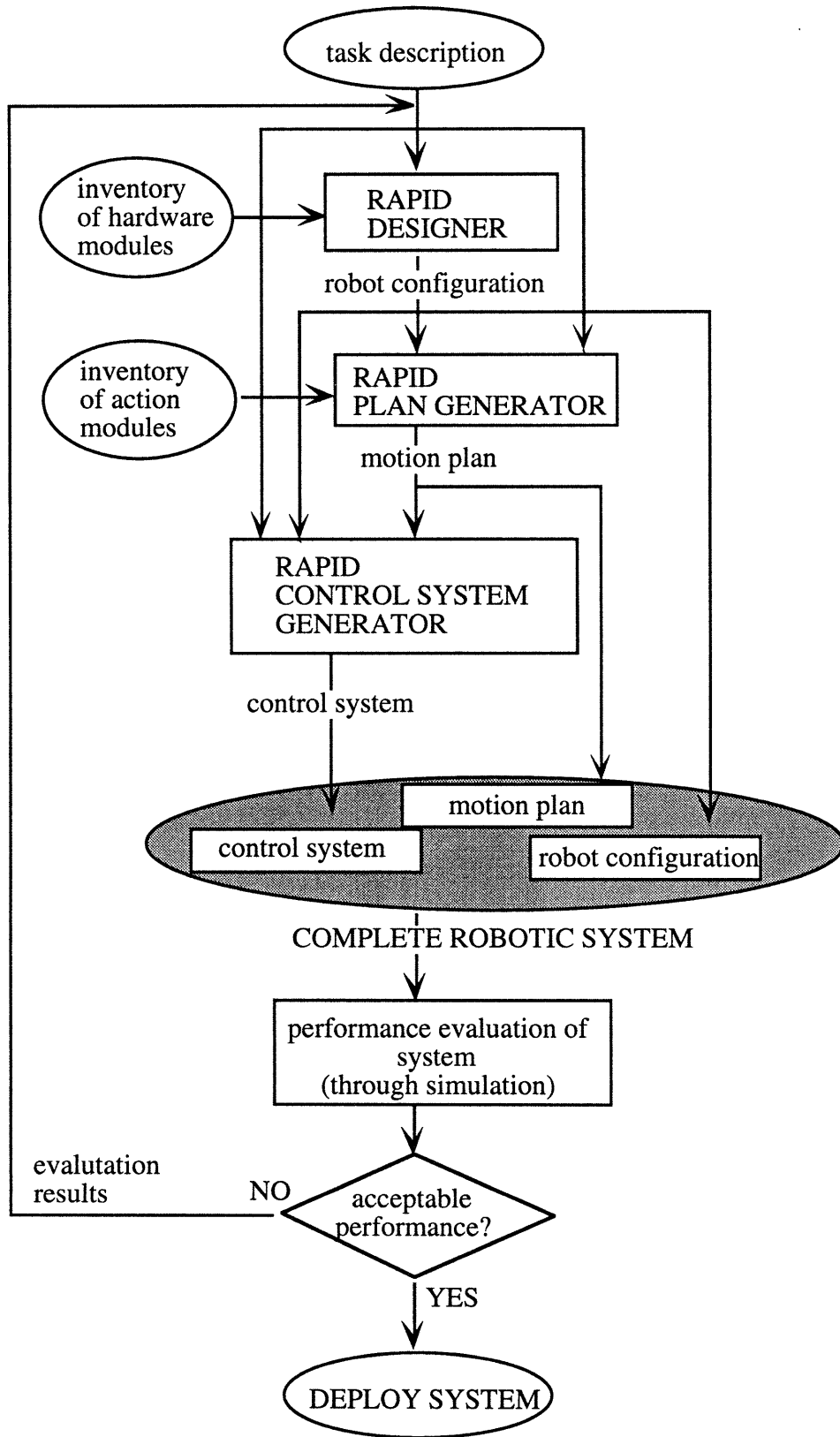
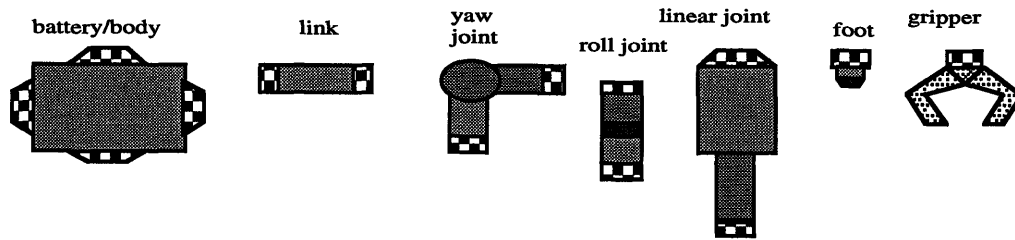


Figure 4 The rapid deployment system design cycle

Sample modules



Sample modular robots

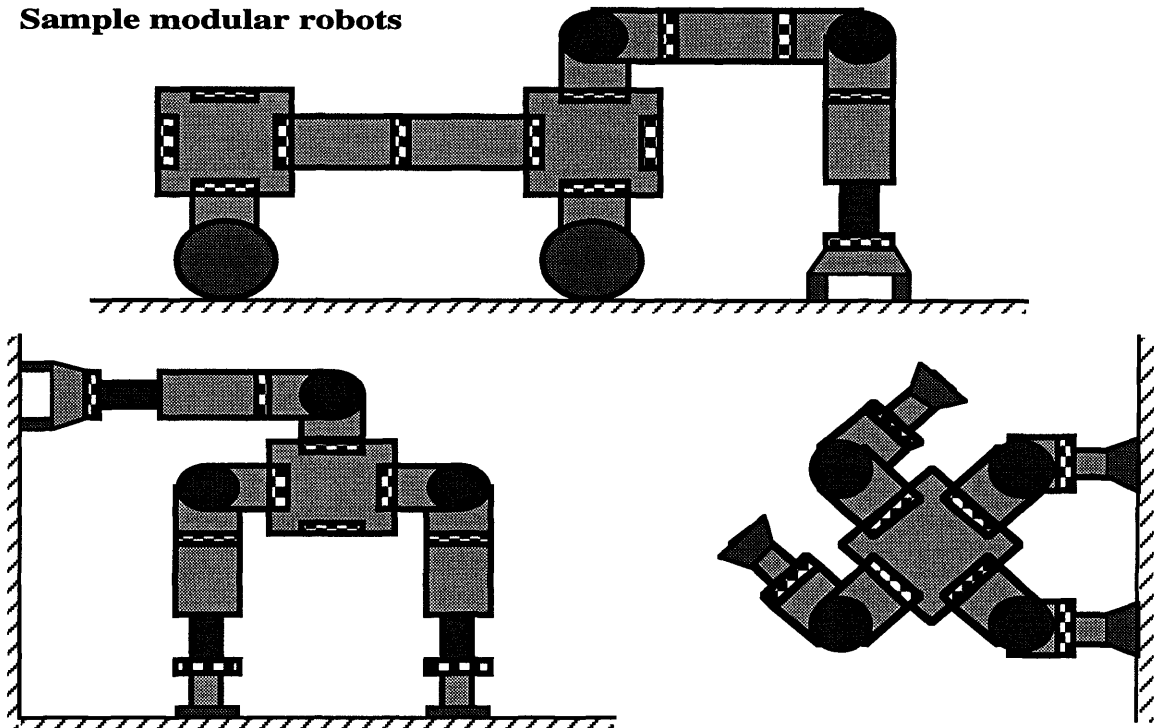


Figure 5 Some example modular design components and assemblies (courtesy [28])

In order to implement this automated search, the user needs to define the criteria for evaluating each configuration. For some tasks, criteria such as power efficiency or speed of task completion make some robot configurations preferable to others.

Even with an inventory of only a small number of modules, the number of possible robot assemblies makes a thorough search of the design space impractical [28]. However, not all of the design space needs to be searched, as task constraints (for example, power constraints or size limitations) preclude the use of some assemblies. Rutman [28] represents these task constraints as “filter rules” which reduce the design space to a size manageable by a typical optimization algorithm. The filter rules can be constructed from physical analysis of the systems involved, as well as from designer experience.

2.3 The rapid plan generator

The approach to rapid plan generation assembles a motion plan from a finite inventory of modules. These modules are called “action modules,” and contain one or more actuator commands. The sequence of action modules comprises the motion plan, termed here as a “script.”

An action module might contain commands to rotate a particular robot joint, apply a force with a specific end-effector, or translate the system center of mass to a new position. For the robot illustrated in figure 6, a partial inventory of action modules is shown in figure 7. Figure 7 also demonstrates one possible script that may be generated from these modules.

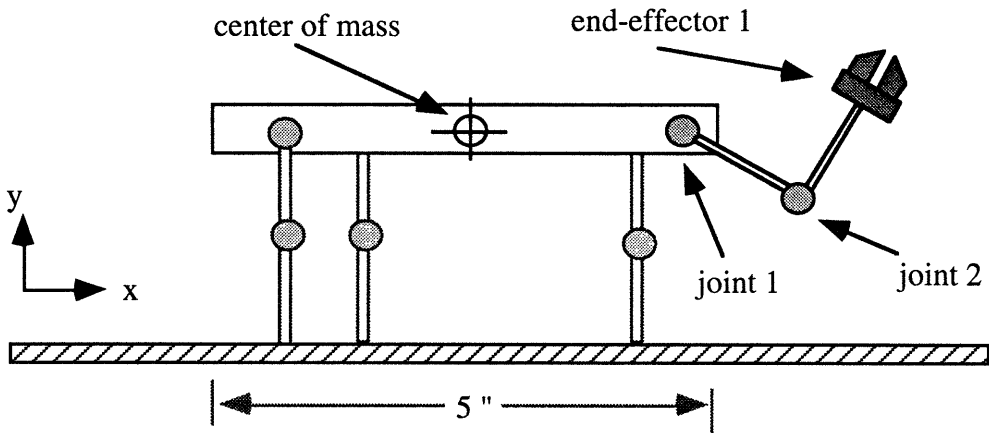


Figure 6 An example of a modular robot

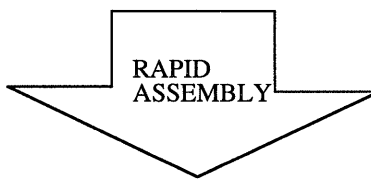
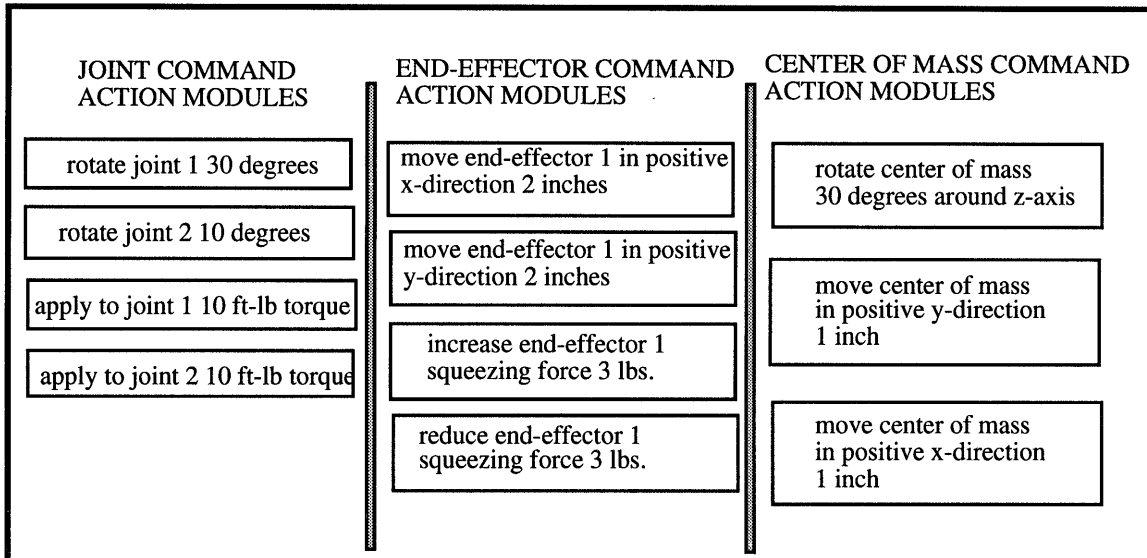
The advantage to this approach is that no custom software must be written for each application. Instead, a motion plan is created by finding a successful assembly of action modules. This procedure is largely automated through the use of a genetic algorithm-based search technique.

The search can be terminated when a script has been found which demonstrates a user-defined level of success. However, in this thesis, the genetic algorithm search technique will be said to search for an “optimal” script.

2.3.1 Creating the inventory of action modules

Selection of an appropriate inventory of action modules is crucial to the success of the rapid plan generator. Including too many superfluous modules may increase solution generation time to an impractical length. On the other hand, including modules which have specific relevance to the task at hand may speed up generation time.

INVENTORY OF ACTION MODULES



SCRIPT

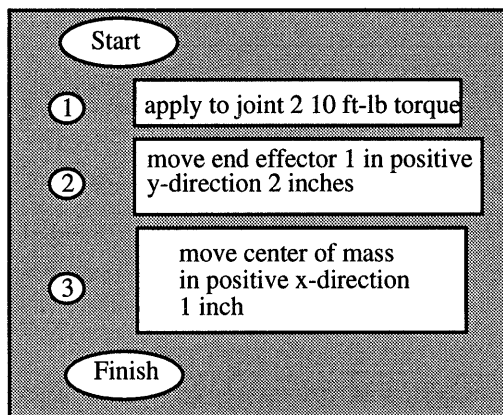


Figure 7 Examples of action modules and a script

There are several constraints on the types of action modules that may be included in the inventory. The action modules used should not violate the robot's kinematic or power constraints. Also, the action modules should not contain robot commands that cannot be executed by one of the control systems available from the rapid deployment system.

Two approaches to inventory selection will be suggested in this thesis. One approach is to construct a set of experience-based rules which select appropriate action modules based on task characteristics. The other approach is to provide an inventory of low-level action modules, then have the rapid plan generator assemble these into useful combinations for the task at hand.

2.3.2 Optimizing the script

For each task, the optimal assembly of action modules is found via the genetic algorithm search technique. Genetic algorithms work well for optimizing computer programs (such as robot motion plans) because of their robust performance in discrete search spaces with undefined derivatives and continuity.

In essence, genetic programming uses the fundamental principles of genetics and evolution in order to produce more and more successful solutions, in this case, action scripts. Genetic algorithms evaluates each solution, and selects the most successful available to pass along its “genetic information” to a new generation of solutions. Some of these “offspring” solutions will be more successful than the parents. In this way, evolution produces a constantly improving pool of solutions.

This procedure is largely automated. However, the performance criteria must be identified, and incorporated into a fitness function. The fitness function is the measure by which the genetic programming algorithm evaluates each script. The fitness function can be constructed by using experience-based rules to identify the appropriate fitness criteria for the given task characteristics.

A potential concern with this approach is that because the scripts are generated off-line beforehand, they will not be robust enough to tolerate the environmental variability actually encountered during the task operation. This thesis presents an approach to training the scripts to succeed for the entire range of predicted environment uncertainty. Simply, the rapid plan generator trains the scripts to succeed for the case of maximum environmental variations. Trained on these cases, the scripts demonstrate robust performance for cases of environmental variation within the predicted range.

2.4 The control system generator

The purpose of the control system is to control the force and positions of the robot body and limbs in order to execute the actions dictated by the plan. A control system consists of a suitable set of generalized control variables, control gains, and control algorithm.

The role of the control system generator is to consider the application task and robot configuration and construct an appropriate control system. Some control systems are too computationally intense to be safely used for some tasks. Some control systems are ill-suited for robots which feature complex mass-distributions or certain classes of degrees-of-freedom. Finally, some control systems require a great deal of sensor information, such as position sensors or accelerometers, which may not be available.

Work needs to be done in this area before a fully-functional rapid-deployment robotic system can be implemented.

Chapter 3 Rapid motion plan generation using genetic programming

The approach of the rapid plan generator is to assemble motion plans from an inventory of action modules. For each task, an optimal assembly of action modules is found using the genetic programming search technique.

Genetic algorithms are employed because of their robust performance in complex, ill-defined search spaces [7]. Genetic algorithms work well for optimizing computer programs (such as robot motion plans). Applying genetic algorithms to the optimization of computer code is known as genetic programming.

This chapter describes how the genetic programming algorithm works. This chapter also describes how to employ genetic programming for the optimization of robot motion plans assembled from action modules.

3.1 Why use genetic programming?

The role of an optimization algorithm is to search a defined solution space for the optimal solution to a specific problem. Most optimization techniques accomplish this by selecting some point in the search space, then using deterministic rules to predict which direction in the search space is most likely to yield a better solution. These rules can be based on such information as search space continuity, modality, or derivatives [7]. The gradient descent algorithm is an example of such a method.

These conventional optimization techniques work well for simple, well-defined search spaces. In many applications, however, the search space is highly-dimensional,

discontinuous, or non-convex. Sometimes, the search space properties are not even known. The conventional optimization algorithms all break down for these applications.

Genetic algorithms were recently developed for optimization in these complex and/or ill-defined search spaces [9]. Genetic algorithms are based on the “biological optimization” technique of genetics and evolution, which improves the survivability of a biological species. These algorithms only use information about each solution, rather than information about the search space. Since these algorithms do not analyze the search space, they require no detailed knowledge about its dimensionality, continuity, or derivatives. Ignoring search space characteristics lends the algorithms computation simplicity, which in turns allow them to more thoroughly cover the search space compared to other techniques.

One optimization problem which has proven difficult until recently is the optimization of computer code. Optimizing computer code requires search among an almost limitless permutation of assembled computer instructions. The search space for this application is highly-dimensional, non-linear, and discontinuous [19,21]. Some attempts were made to optimize computer code through conventional search techniques, but with limited success [21]. Genetic algorithms, however, were a natural and successful choice for traversing the complex search space required to optimize computer programs. The application of genetic algorithms to optimizing computer code is known as genetic programming, and has been used for such purposes as optimizing sorting algorithms [20] and creating neural nets [7].

Robot planning is essentially the task of creating a computer program which controls robot activity. The program is an assembly of instructions which dictate actuator movements. Soon after genetic programming was proven successful, it was applied towards the automatic assembly of robot plans [21, 25, 26]. These experiments proved successful for the simple robot and environmental models used.

3.2 The genetic algorithm

The mechanism of genetic algorithms is grounded in the biological principles of genetics and evolution. In biological evolution, only the strongest members of a species survive. These members then reproduce, yielding offspring who inherit a combination of the parents’ genes. Because an offspring’s genes are inherited from both parents’, the offspring exhibits a blend of the parents’ features. Ideally, the new features represented by these genes retain the successful qualities of the parents, or demonstrate an improvement over the parents. If so, the offspring live to reproduce and pass along the traits. If a new

trait is a liability to survival, the offspring is unlikely to live until reproduction, and so the trait does not get passed on. In this way, as these generations of reproduction continue, the species as a whole becomes stronger.

The genetic algorithm search technique exploits these biological principles of genetics and evolution to produce improved “generations” of solutions to a specific problem. Each solution contains “genes”, or pieces of information which describe one part of the solution. The solutions which best solve the specified problem are selected by the genetic algorithm for reproduction. These solutions are then paired up into sets of “parents”. The genes of the parents are combined to produce new, offspring solutions. As in biological evolution, the new generation of solutions will feature characteristics of the parents, but in a slightly different manner. Some of the offspring solutions will be better than the parents, some will be worse. Only the best ones will go on to have offspring of their own. In this way, each generation of solutions improves over the last.

Each iteration in the genetic algorithm consists of three phases: evaluation, selection, and crossover. The genetic algorithm cycle is illustrated in figure 8. In the evaluation phase, each individual solution is assigned a score based on some user-specified fitness function. In the selection phase, the highest-scoring individuals are selected for reproduction. Finally, in the crossover phase, or reproduction phase, offspring solutions are created which inherit the genetic information contained by its parents. Then, the cycle begins again by re-evaluating all of the solutions.

Example

The following example (illustrated in figure 9) explains how a solution to a specific problem may be described by a set of genes, and how this genetic information is passed along to offspring solutions. This example is intended to be an overview to the genetic algorithm cycle. The genetic algorithm phases of evolution, selection, and crossover are discussed in detail in the remainder of this chapter.

Consider an application to optimize the volume-to-surface-area ratio of a cardboard box. Each solution might represent an individual box’s length, width, and height. A gene string to represent this solution might consist of three genes, one to define each of these parameters. A gene string for one box might be [5,3,3], representing the box’s dimensions of 5 inches long, 3 inches wide, and 3 inches tall. A gene string for a different box might be [2,5,4], representing dimensions 2 inches long, 5 inches wide, and 4 inch tall. Both boxes have a gene string similar in form. The difference in the box’s features is due to the difference in the boxes’ respective genes.

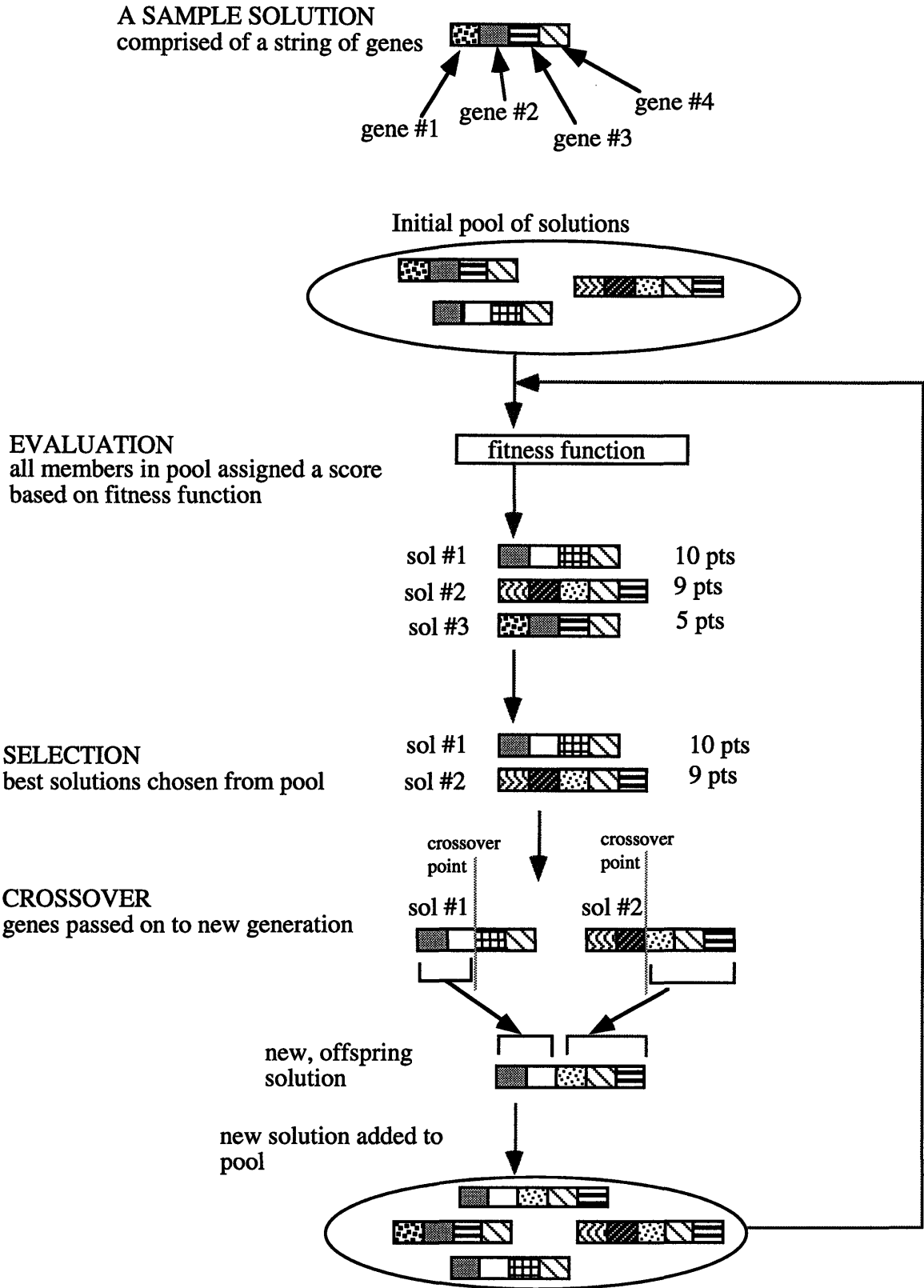


Figure 8 The genetic algorithm cycle

The genetic algorithm selects the most successful solutions for “reproduction”, or crossover of genetic information. The first box would have a volume-to-surface-area ratio of .57, the second box .53. Suppose that these two boxes represent the best solutions available thus far. These two boxes would then be selected for reproduction.

The offspring of these two boxes would contain genes inherited from its parents. For example, one possible offspring might contain a gene string of [5, 5, 4]. This offspring box inherits its length (first gene) from the first parent, the width (second gene) from the second parent, and its height (third gene) from the first parent. The offspring box’s characteristics are a combination of the parents’ characteristics. The offspring box, however, has a volume-to-surface-area ratio better than either of its parents: .77. In this case, the blend of genes gives the offspring box improved characteristics. The box’s dimensions are all very similar in magnitude, so it is very “cube-like.” A cube represents the optimal volume-to-surface area box configuration.

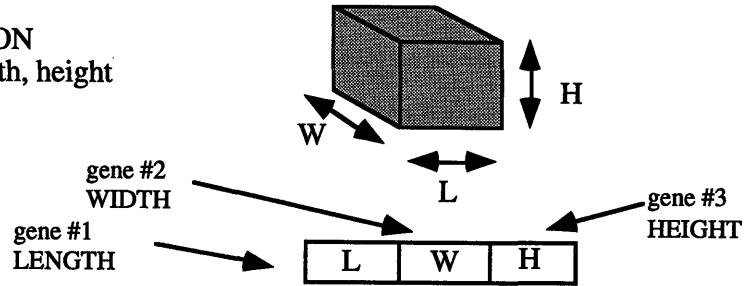
This box now represents the best solution thus far, and stands a good chance of being selected for reproduction. In this way, the offspring’s useful “cube-like” traits will perpetuate through the following generations. If the inherited gene produced poor traits, the box would not be selected for reproduction, and the trait would die out of the gene pool.

The following sections describe in detail how genetic algorithms can be applied to rapid plan generation. Specifically, the form of the genes and the mechanisms of crossover and evaluation will be discussed.

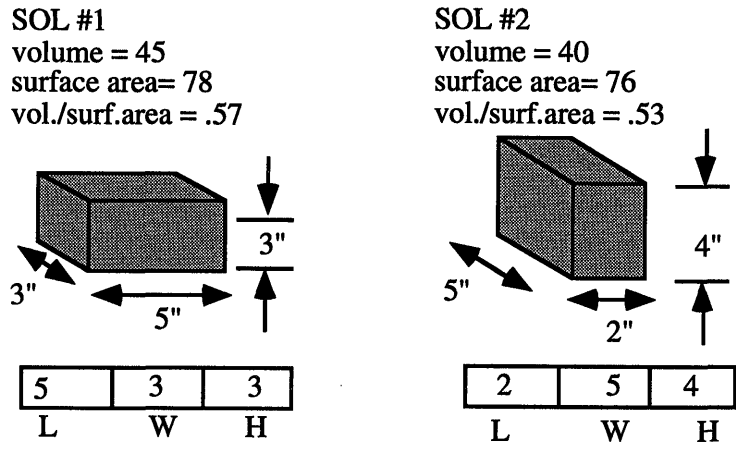
3.3 The genes for robot motion plans: “action modules”

As discussed, genetic algorithms work with solutions comprised of discrete “genes.” These genes contain information defining one aspect of the solution. In the box example, a gene was used to define each box dimension. In an application to create a neural net, one gene might define the number of layers of nodes, while another gene might define the number of nodes in each layer. Each individual is unique because of its unique genes; for example, one individual box may have “height gene” that dictates a very large height, while another may have a height gene that dictates a very small height. Because the respective genes are different, the boxes have different sizes.

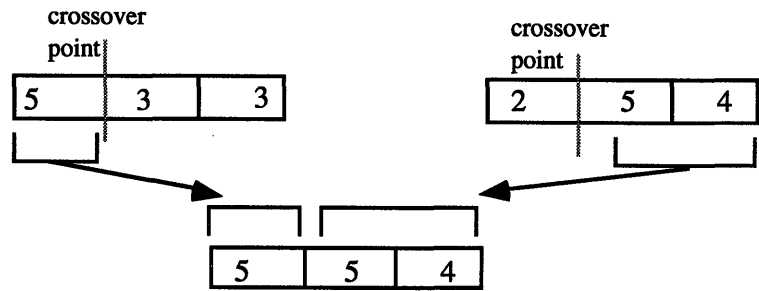
A SAMPLE SOLUTION
a box with length, width, height



TWO ACTUAL SOLUTION
represented by gene strings



CROSSOVER
genes combined to produce
new gene string



A NEW, OFFSPRING SOLUTION
represented by the new gene string
In this case, the offspring is better than
the parents (better vol/s.a. ratio)

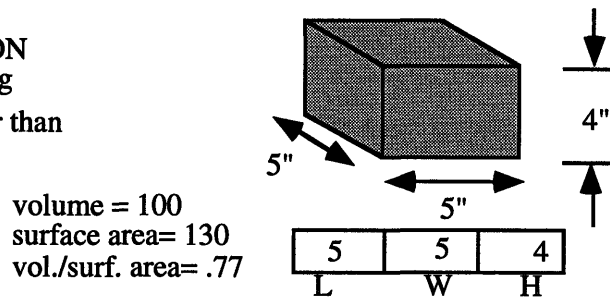


Figure 9 An example optimization problem using genetic algorithms

In order to apply genetic algorithms, one must decide how each solution will be described by genes. There is no formal methodology for doing this. Furthermore, a variety of genes could be used to solve the same problem. For example, in the neural net problem, an alternative gene string could include a gene which defined the total number of nodes to be

used in the net, and a gene which represented the number of layers to be used. This thesis describes one approach to defining a robot motion plan via genes, within the context of modular robotic systems.

3.3.1 The content of an action module

In optimizing robot motion plans via genetic algorithms, each solution is a motion plan which the robot executes to accomplish a specific task. A complete plan contains the sequence of actuator commands necessary for the robot to accomplish the task. This motion plan must be described by discrete genes.

For optimizing robot motion plans, the following gene format is proposed. Every gene is an instruction, or series of instructions, that commands robot action. The concept of scripts being represented by gene strings is shown in figure 10. A gene might instruct the robot to move a particular joint by 30 degrees. Another gene might instruct the robot to lower the height of its center of mass by 2". Within the context of modular robotic systems, these genes are referred to as an "action modules," each with its own unique robot motion control instructions.

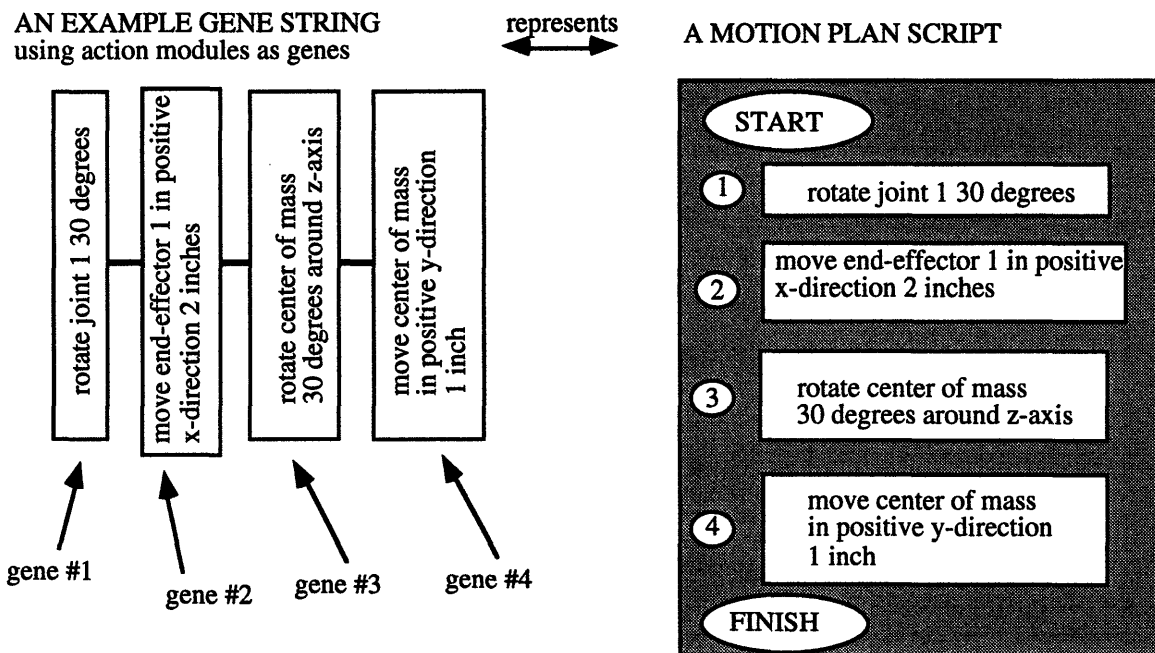


Figure 10 Action modules as genes

Each solution is initially composed of a string of 10 action modules (for purposes of clarity, the example in the diagram shows string of four). Within the context of modular robotic systems, this string of action modules is referred to as a “script.” Action modules might be repeated several times in a script. The sequence of action modules in the script determines the order that the robot executes them. When the robot reaches the end of script, it stops executing instructions.

Modules can be created which lengthen the script size. Such a module is described in chapter 4. This module, called “Do 3”, lengthens the script by three action modules. Since any number of “Do 3” modules can be included in the script, the maximum script length is unbounded.

Each script represents one solution to the problem. When these scripts reproduce, the action modules are the genes that get passed along the offspring scripts. Thus, the offspring scripts are motion plans composed of some combination of the parents’ action modules.

There are some constraints on the content of an action module. Any action module should be comprised of direct actuator instructions that can be implemented by a control system. Most control systems are capable of doing the following:

- moving any point on the robot to an arbitrary position in inertial space.
- moving any point on the robot by some amount relative to its current position in inertial space.
- controlling the velocity of any point on the robot
- controlling the acceleration of any point on the robot
- controlling the interaction force between the environment and any point on the robot.

Additionally, the robot instructions are constrained by robot kinematics and power capabilities. A robot module should not instruct the robot to move into a position outside of its kinematic workspace or power workspace. If a script contained an action module that violated these constraints, the robot might fail. A script leading to robot failure would receive low fitness points (see section 3.4).

One approach to excluding action modules which violate task constraints is to use rules to select appropriate modules based on task characteristics. These rules may be derived from

physical analysis or user experience. This rule-based approach is described in section 3.3.2.

Within the discussed constraints, an action module can contain almost any combination of available sensor information and actuator instructions. For example, a module might instruct the robot to “reduce the squeezing force of end-effector 4 by 50%”, requiring some sensor processing as well as motor command.

An action module might contain a series of several instructions which together comprise some complex activity. For example, a module which instruct the robot to take a step forward actually contains a string of several commands: “move end-effector 1 in the positive z-direction (in inertial space) 2 inches; next, move end-effector 1 in the positive x-direction 1 inch; next, move end-effector 1 in the negative z-direction 2 inches.” In general terms, this action module is equivalent to “lift leg; move leg forward; lower leg.”

A module may also combine sensor information and logic to specify the conditions under which a command should be executed. For example, a module might instruct the robot that “if the contact sensor number 2 is activated, move joint 3 negative .3 radians; otherwise, move joint 4 .1 radians.” Incorporating logic into the action modules allows the script to flexibly respond to a variety of possible expected environmental conditions. If a certain set of sensor criteria are met, one set of instructions is executed; otherwise, a different set of instructions is executed.

While these logic-based modules allow plan flexibility, the script format hinders the speed with which the plan reacts to changing environmental conditions. In the script format, the robot executes each action module in series; when one action module has been executed, the robot moves on to the next one. The drawback to this approach is that sometimes the user wishes the robot to be executing one set of instructions, but to constantly monitor some specific sensor information to trigger a change the executed instructions. For example, it might be desired for a robot to walk forward until hitting a wall. To accomplish this using the script format, a module which checks for wall contact (through a contact sensor) would have to be interspersed throughout the script so that the sensor information would be repeatedly checked.

The approach using logic-based modules proves successful (see chapter 5) for the class of problems addressed by this thesis, in which the environments feature a small range of dimensional variation from the nominal case. However, a different class of unstructured environments may feature “topological” changes, such as the presence of unexpected obstacles or major environmental variation. In these cases, a better plan structure would

evaluate all sensor information simultaneously, and “branch” to a correct “sub-script” based on the sensor processing and some rules.

Subsumptive plans prove robust enough to tolerate topological variation in the environment [22, 23, 24]. Subsumptive plans execute modules in parallel, rather than in series. In this way, the plan may branch to the appropriate set of robot commands at any moment. Subsumptive plans have been successfully applied to robot control [22, 23, 24], and have been rapidly generated through genetic algorithms [21].

It has been demonstrated in genetic programming that individual modules can accept arguments [21]. Such a module would be very useful in generating robot motion plans. For example, a module instructing the robot to squeeze an object with its end-effector might accept an argument dictating the magnitude of force that the robot squeezes. In this way, a motion plan could instruct the robot to squeeze with a range of forces using only one “squeezing module” and a mechanism for passing a range of arguments. The more impractical alternative is to have one squeeze module for every possible desired force. Using arguments in genetic programming for robot control is successfully demonstrated in [26].

3.3.2 Creating the inventory of action modules for a specific robot application

Some methodology is needed to derive an inventory of modules for each specific robot application. This section describes two possible approaches for this task.

Rule-based selection

Robot kinematics and power characteristics, environmental features, and task characteristics all require a minimum set of action modules, and preclude the use of others. One approach to creating the inventory of modules is to describe these requirements by a set of rules. Using this approach, each application (robot, task and environment) maps to an inventory of available action modules.

One rule to create an inventory of action modules would require two modules for each of the robot’s degrees of freedom. One module controls the angular displacement of the joint’s rotation (e.g. a “rotate joint 1” module), another module controls the torque output by the joint. Note that position and force control of a joint may not be executed simultaneously. This conflict will not arise within the script format, as only one action module is executed at a time. With this inventory of modules, the genetic algorithm has the

freedom to control the displacement or joints of all modules on the robot. The range of displacement or torque commanded by these modules is constrained to robot capabilities. Such tools as the joint force-workspace map [29, 30, 31, 31] or the power map [32] may be incorporated into the selection process to determine which modules are unusable.

In addition to this rule, one could create rules which include commands useful for specific applications. For limbed robot locomotion, balance and stability are maintained by controlling the location of the robot's center of mass relative to its positions of environmental contact. For applications in which it seems likely that the robot will use its limbs for locomotion, a module should be included which controls the displacement or velocity of the center of mass.

For some applications, motion plan generation would be facilitated by including in the inventory action modules which define often-repeated sequences of actions. In the limbed robot locomotion example, the robot will be required to repeatedly take steps with each limb. This stepping action is composed of lifting the limb, then moving it forward, then lowering it. Modules could be included in the inventory which dictate the instructions for a limb to take a step, so that the genetic algorithm does not have to repeatedly assemble the instructions to execute this action.

In implementation, the selection of appropriate action modules can be facilitated by such tools as the joint workspace map. However, many useful rules will be derived from user experience as to what types of modules are appropriate for each application.

It should be noted that it is important not to include modules which contribute little to the success of the solution. While it is true that the genetic algorithm may "weed out" the useless modules from the solution, this may come at the expense of increased genetic algorithms cycles, and hence longer solution generation time. One may construct these "filter rules" (as in [28]) based on experience.

Using Automatically Defined Functions (ADF)

Another approach is to have the genetic algorithm itself identify which modules are relevant to a specific task. This technique of using "Automatically Defined Functions" (ADF) was pioneered by [18]. The advantage to this approach the inclusion of good modules is not limited to the user's expertise.

In order to implement this approach, the genetic algorithm must be provided with an initial inventory of modules that are basic ("low-level") enough to apply to any of the anticipated

applications. As the genetic algorithm assembles a script to a specific problem, sequences of these “low level” modules will recur throughout the script. After a specified number of iterations, the genetic algorithm will review its script for these repeated sequences. When one is found, the genetic algorithm creates a new module which contains these repeated instructions (this module is called an ADF). The new module is then added to the inventory. Using the new inventory, the genetic algorithm continues assembling the script.

The premise of this approach is that repeated sequences of instructions are useful towards completing the task. With the newly-created “high-level” blocks, the genetic algorithm is relieved from having to “evolve” a (possibly very large) set of instructions every time it is needed. Instead, one module does the same job. Using ADF’s has proven to reduce the time required to generate a solution for some problems [25].

It is interesting to note that some sequence of these high-level blocks may, in turn, appear repeatedly throughout the script. The genetic algorithm may construct even “higher-level” blocks containing a sequence of the high-level blocks. In fact, this process can be repeated as often as desired to create modules of extremely high levels of abstraction [18].

For a rapid motion plan application, the low-level modules would have to be applicable regardless of the task or environment. Such modules might control only the degrees of freedom of the robot, without stringing together any sequences of these commands. The manner in which these modules could be selected based on robot configuration has already been suggested in this section. Such modules might instruct the robot to displace the end-effector, or apply a force with a given end-effector. Some tasks might require intricate sequences of instructions to be repeated, such as taking a step with one of the limbs. This often-repeated action would be a good candidate for the genetic algorithm to incorporate into a higher-level ADF.

3.4 Evaluating a script

Each string of action modules, or script, represents one solution to the given problem. The genetic algorithm picks the successful solutions and passes a combination of their genetic information on to a new generation of solutions (see next section for description of selection process). Consequently, some measure is needed for evaluating the relative success of each solution.

This evaluation is done via a utility function, or “fitness” function. The fitness function is the rule by which a performance score is assigned to each individual solution.

For a rapid plan generator, the fitness function evaluates the robot's performance of a specific task based on some user-criteria. The most practical way to evaluate the robot's performance is through off-line simulation.

The criteria for task success is completely subject to the user's discretion, and usually changes from application to application. For example, in an application where a robot is required to put out a fire in a remote forest, speed of task accomplishment is a critical measure of success. On the other hand, for a robot required to autonomously survey a large parcel of land, power consumption is a very important factor. Usually, the task success is some combination of several criteria.

Somehow, the task evaluation criteria should be assembled together into a quantifiable function for robot performance evaluation. As in defining any type of utility function, this is not a straightforward task. The user must determine the best quantifiable measure of each of these criteria. Then he must decide how these measures will be weighted relative to each other in the overall function.

Creating fitness functions for genetic algorithms presents special difficulties. Often the user's attitude is that success is binary--that is, either the robot was successful, or it wasn't. However, the genetic algorithm needs more information than binary success; it needs to know the relative worth of every solution so that it can pick the best for reproduction. This means that the user must invent a useful measure for determining which "unsuccessful" solutions were better than others. This concept of assigning "partial credit" is essential to a useful fitness function.

For example, consider the simple robot task illustrated in figure 11, where the goal is for a limbed robot to walk from the "Start" point to the data collection box "Goal" without falling. Assume that motion plans can be constructed to generate locomotion.

In writing a fitness function, the user might be tempted to assign a maximum score for reaching the Goal, and a zero score for not reaching the Goal:

$$\begin{aligned} \text{fitness} &= 100 \text{ points (if reached Goal)} \\ &= 0 \text{ points (if not reached Goal)} \end{aligned}$$

Using this approach, a robot that walked most of the way to the goal before falling (as in figure 11) would receive a score of 0. However, the motion plan shows promise; the robot traveled most of the way to the goal. Somehow, this solution should be awarded some

fitness points to give it a chance to pass along its genes. A better fitness function would assign a score based on how far the robot traveled towards the goal:

$$\text{fitness (points)} = \text{distance traveled (inches) toward goal}$$

A maximum score would be 100 points, reflecting a robot that traveled the entire 100". The closer a robot gets to the goal, the more points it is awarded. The robot shown in figure 11 receives 85 points. The robot motion plan has a very good chance of being selected to pass along its genes.

After many genetic algorithm cycles, many generated solutions may evolve which demonstrate undesirable characteristics. For example, consider two robots shown in figure 12 which are trying to perform the same task as in figure 11. Robot 1 takes a very direct route towards the goal, but falls just before reaching it. Robot 1 travels 85" and expends 4J of energy before falling. Robot 2 takes a more circuitous route toward the goal, getting only slightly closer than Robot 1. Robot 2 travels 90" before stopping (ran out of energy), and expends 10J of energy.

Using the same fitness function, which awards points based on travel distance only, Robot 2 scores better than Robot 1:

$$\text{fitness points} = \text{distance traveled towards goal (inches)}$$

$$\text{Robot 1 fitness} = 85 \text{ points (for 85" traveled)}$$

$$\text{Robot 2 fitness} = 90 \text{ points (for 90 " traveled)}$$

The user might decide that the route taken by Robot 2 is far too inefficient to be of any value. It would then be desirable for Robot 1's motion plan to pass along its genes, and not Robot 2's motion plan.

In that case, a new fitness function must be defined which incorporates the new performance criteria:

$$\text{fitness} = \text{distance traveled (inches)} - 5 * \text{energy spent (J)}$$

This fitness function awards points for distance traveled, but deducts points for wasting energy. In this case, the energy deduction is multiplied by a weight of 5, since the scales of possible distance traveled (85) and energy spent (10) is different. In a real application, the weight is determined by the relative scales of the fitness function criteria, and how important each criteria is relative to each other.

With this fitness function, Robot 1 now receives a better score than Robot 2:

Robot 1 (85" traveled, 4J spent) fitness= $85 - 5 \cdot 4 = 65$ points

Robot 2 (90" traveled, 10J spent) fitness= $90 - 5 \cdot 10 = 40$ points

Robot 1 now becomes the best candidate for reproduction. The revised fitness function accommodates the unexpected and unwelcome evolution of energy-efficient solutions.

Clearly, it is undesirable for the user of a rapid plan generator to monitor the evolution of the solutions to identify these undesirable directions of evolution. One solution to this problem is to assess the effect of various fitness function criteria on the generated solutions for many types of problems before actual application. Rules can then be made which identify critical fitness function criteria for specific task characteristics.

The off-line evaluation approach presents an additional problem which must be addressed before implementing a rapid plan generator. It is undesirable to create a new simulation environment for every new task application. A true rapid-plan generator requires the use of a general-case simulator that can accommodate a wide variety of robots, environments, and tasks.

Because each cycle of the genetic algorithm requires simulation to evaluate a solution, there is a trade-off between the complexity of the simulation model and time required to generate a solution. Currently, no work has verified whether simulation models are accurate enough such that genetic algorithm-generated robot motion plans are accurate enough to apply to actual hardware applications. However, based on the results of this thesis (see Results chapter), it is expected that today's computational technology is quick enough to generate usable robot motion plans within a realistic period of time (hours-days) for a rapid deployment system.

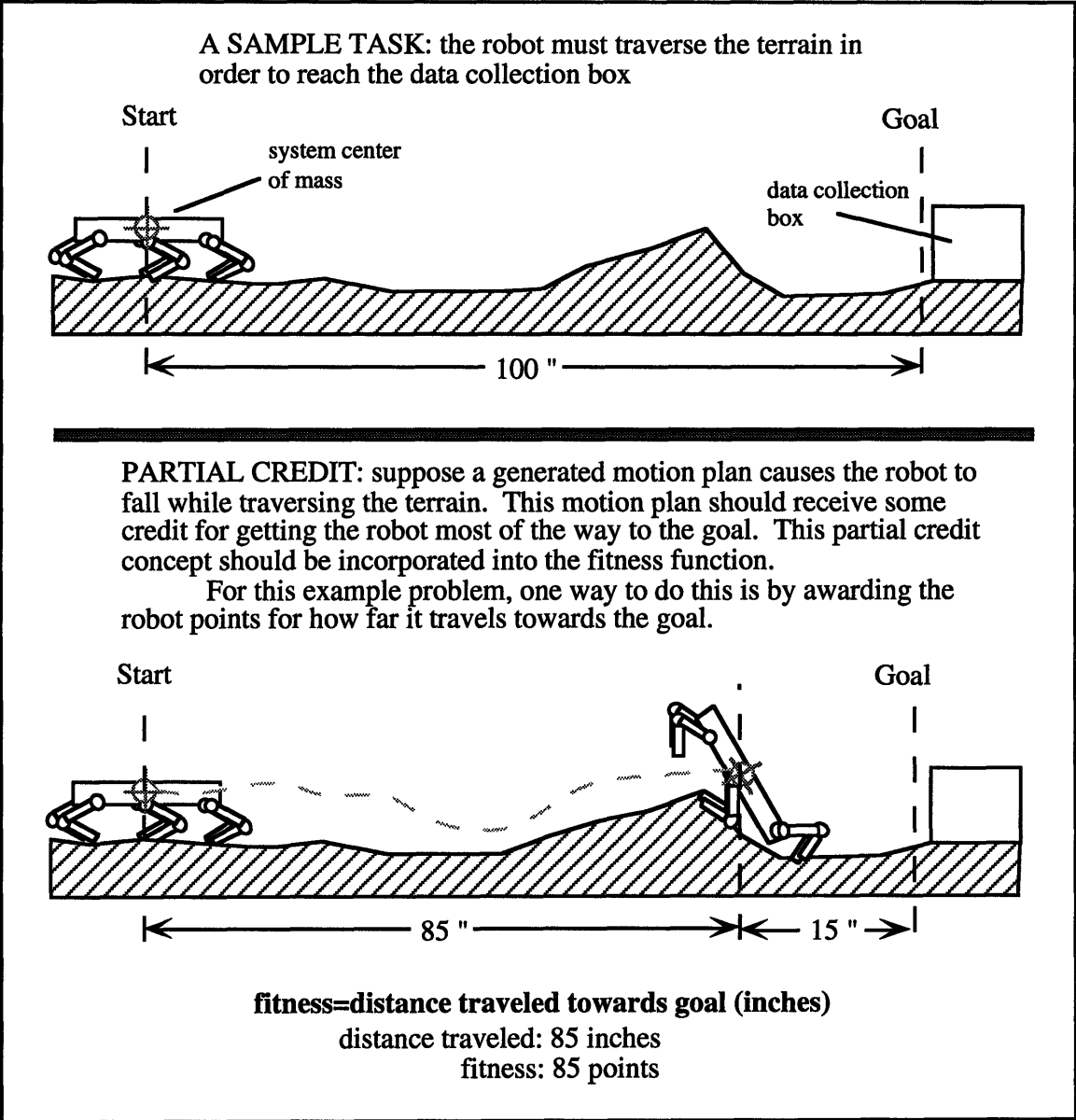


Figure 11 Partial credit in the fitness function

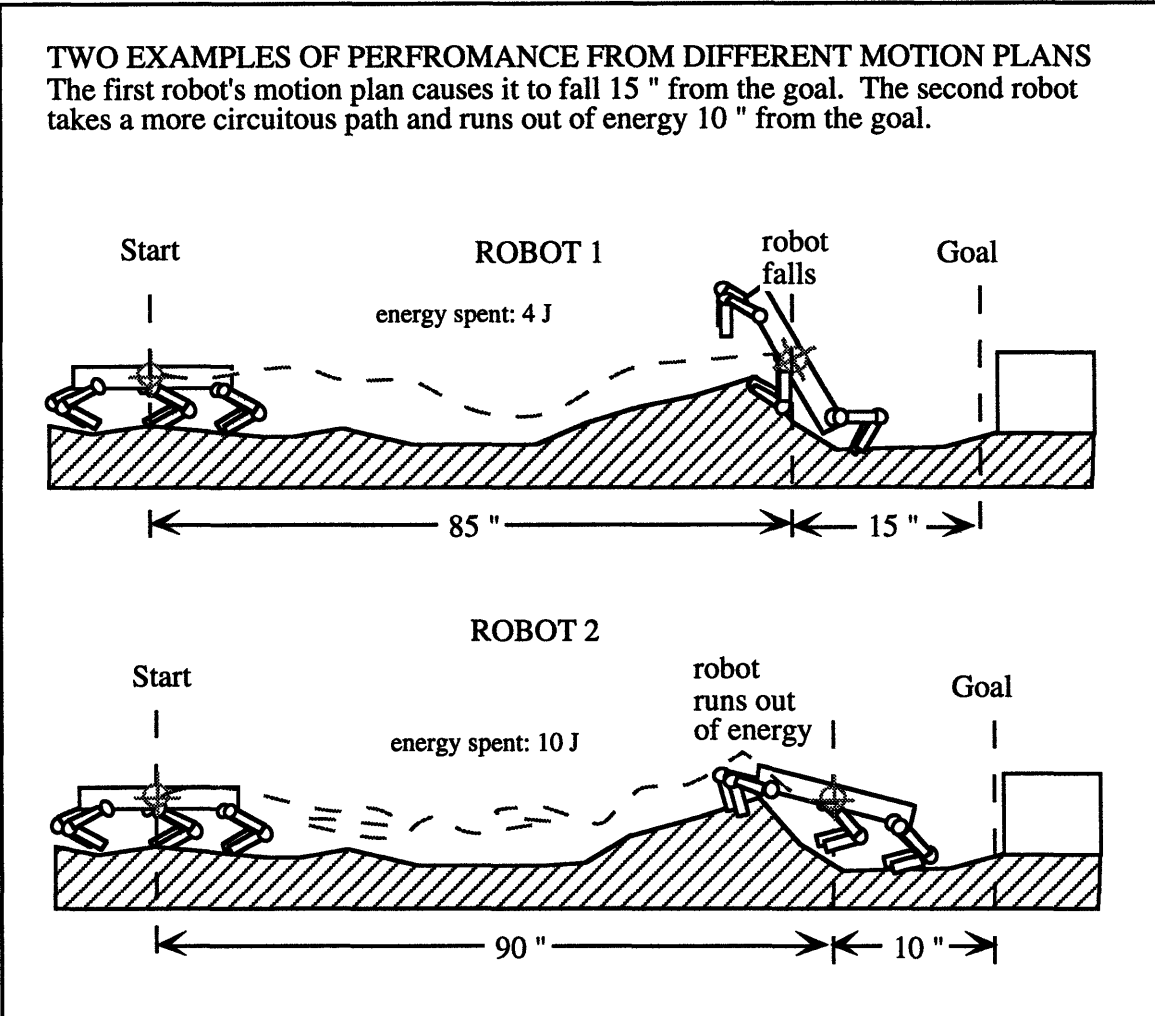


Figure 12 Performance cases which require additional fitness function criteria

3.5 Creating offspring scripts

After a generation of scripts has been evaluated, the genetic algorithm selects some for reproduction. One way to do this is to select a fraction of the best scripts available. However, the script that has the most problem-solving potential is not always one of the best scripts in the early generations. A better approach to picking scripts is to make every script a candidate for reproduction, but weight the better scripts as more likely to be picked. Such an implementation is called "roulette-style" selection [7].

Once the parents are selected, the genetic algorithms must combine these scripts' genes (action modules) to produce new, offspring scripts. The two most common mechanisms for doing this are crossover and mutation.

Crossover

Crossover is the means by which some combination of the two parents' genes are passed on to a child. Just as in biological genetics, it is hoped that the children inherit the successful genes of the parents, but with a slight variation that improves upon the parents' qualities.

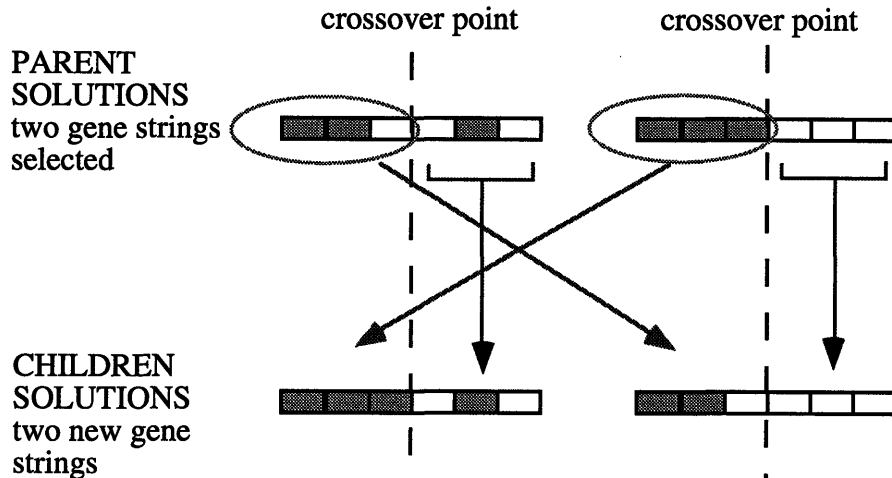


Figure 13 The mechanism of crossover

The mechanism of crossover is simple (see figure 13). The genetic algorithm randomly selects two scripts from among the previously selected “fit” solutions to act as parents. It then randomly picks a “crossover point” in the gene string of each parent. From one parent, the genes on the left side of the crossover point from are combined with the genes on the right of the crossover point from the other parent, to create a complete gene string for a new offspring. A second offspring is created using the genes from the right side of the first parent, and the left side of the second. Now, two new “children” scripts exist which contain all of the genes of the parents, but in a slightly different arrangement.

Mutation

A second mechanism for varying the genetic arrangement of the offspring is mutation. Mutation endows the offspring with genes which are not shared by either parent.

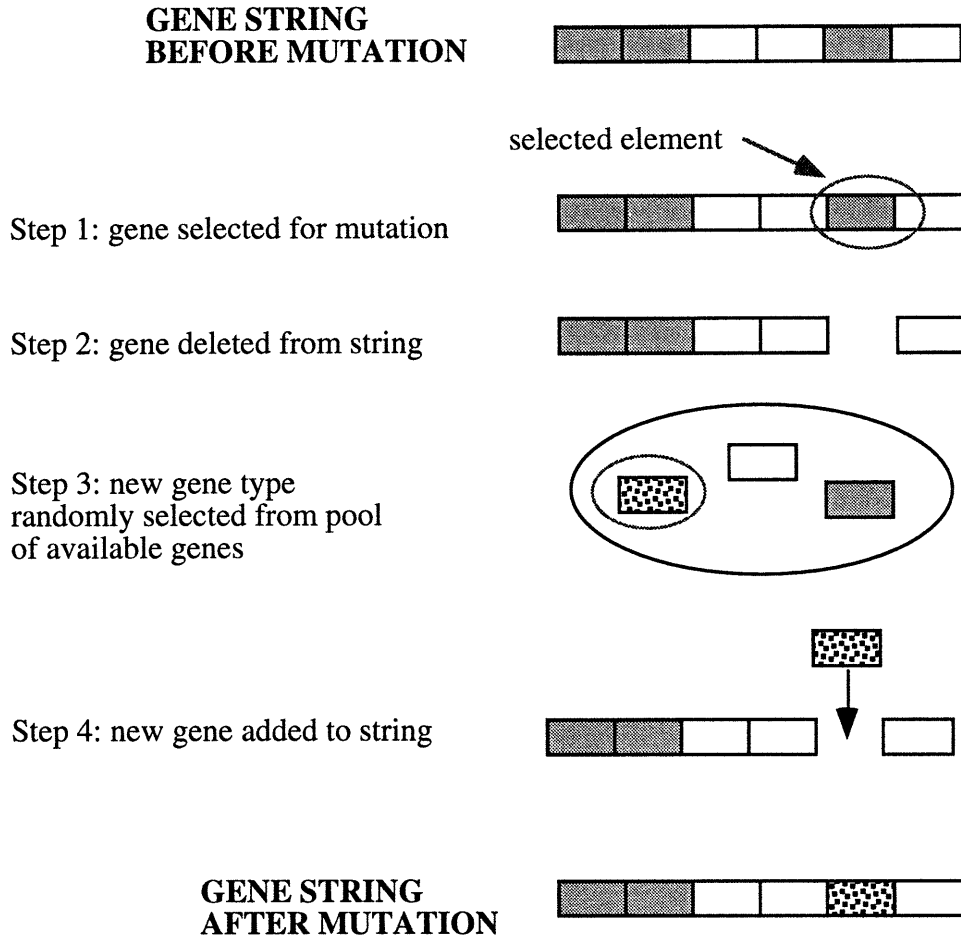


Figure 14 The mechanism of mutation

The mechanism for mutation is also very simple (see figure 14). One gene in the string is randomly chosen and then deleted from the string. The genetic algorithm then randomly picks one replacement gene from among the available genes and inserts it in place of the deleted gene. Now, the child solution is exactly as it was before, but with one new gene that did not come from either parent.

In implementation, some combination of these two mechanisms is used [9]. Crossover by far is the most important mechanism. Crossover allows genes which made the parents successful to be passed along a new generation of solutions. It is hoped that the new arrangement of these useful genes improves upon the already successful arrangement in the parents.

While mutation is critical to introducing new genes into the gene pool, it is usually set to a very low, non-zero rate [18]. High rates of mutation tend to replace genes that made a solution successful, perturbing the solutions from their course of useful evolution.

After crossover and mutation performed, the genetic algorithm cycle begins again, starting with the evaluation of the new generation of solutions.

It is important to note that there is randomness in the algorithm in the selection of parents, the selection of crossover points, and in mutation. Consequently, running the procedure twice generally does not produce the same results. The user may find that some runs find a successful solution more quickly than others, while some get stuck in a rut and never find a successful solution.

3.6 Picking good training problems to improve robustness

Usually, the fitness function evaluates the performance of the script on just one training problem. Thus, the generated solution is not a general-case solution to a specific class of problems, but a specific solution to one specific problem. If a script is used on a problem which differs from the training problem, it generally does not work as well.

In unstructured environment applications, there may be considerable difference between the environmental characteristics known beforehand, and those actually encountered. In the nuclear power plant example, pipes may have unknowingly broken, cracked, or worn down; fungus may have altered the coefficient of friction of some pipe walls.

A robot motion plan for unstructured environment applications must be robust enough to succeed despite these unexpected variations. Clearly, some methodology is needed to train these scripts to accommodate a range of potential variations in the application environment.

One technique for doing this is to have the fitness function evaluate a script's performance on a series of training samples, spanning the range of predicted environmental variability. A successful script accomplishes a specific task over the entire environmental range.

Experience with the application environment helps to predict the range of this environmental variability. The tolerance on the nominal environmental dimensions should be included in the predicted variability.

Some of these environmental variations will adversely effect the success of a generated script, while some will not. A methodology is required to identify the detrimental variations and then train a script to accommodate them.

The following technique for creating robust scripts is illustrated in the example problem in the next chapter. The first step is predict the range of environmental variability, as already described. The next step is to generate a script which succeeds on the nominal-case environment. Then, this script is evaluated on each of the variations predicted. Those variations which adversely effect the script's performance should be identified as "worst-case" variations.

Now that the detrimental variations are identified, a script must be generated which succeeds for all of the worst cases. The fitness function should be changed to evaluate a script's success on all of the worst case samples, and a new script should be generated which optimizes performance for these cases. Results in Chapter 5 indicate that the resulting script may perform robustly over the range of expected application variation.

It should be noted that a more complex class of unstructured environments may feature "topological" changes, or substantial variation to the environmental configuration. These variations require not only robustness to dimensional variability, but frequently a complete change of instructions. This class of problems is not addressed by this thesis.

Chapter 4 A sample application--restoring the USS Constitution

A rapidly deployable modular robotic system is useful for restoration or maintenance tasks in unstructured environments. Robots could accomplish such tasks which would be difficult, expensive, or impossible for humans.

One potential application site is aboard the USS Constitution [33, 34]. Robots could accomplish many inspection and restoration tasks which preserve the ship. One of these tasks is the inspection of the ballast platform planking for rot.

This chapter describes the ballast platform inspection task in detail, and describes how a rapid motion plan generator was constructed to solve the task.

4.1 The ballast decking inspection task

The USS Constitution (illustrated in figure 15) is the oldest fully-commissioned warship in the US Navy. Launched in 1797, "Old Ironsides" has never been defeated in battle. The Constitution plays an important role in US history, and is visited annually by thousands of tourists. Berthed in Boston Harbor, the Constitution requires a great deal of restoration and maintenance to ensure that it does not fall victim to aging and weathering.

One such maintenance task is the inspection of the masts for rot. These masts are roughly 200 feet high, and are nested in yards and rigging. The masts' height makes it dangerous for humans to climb them to inspect for rot. Currently, they are inspected by man using a large crane that is set alongside the ship. The crane is expensive, unsightly, and detracts from the tourists' experience aboard the ship. Also, the rigging makes it difficult for the

inspector to survey the entire mast. Robots might be constructed to scurry up the masts, inspecting them more quickly, safely, and aesthetically.

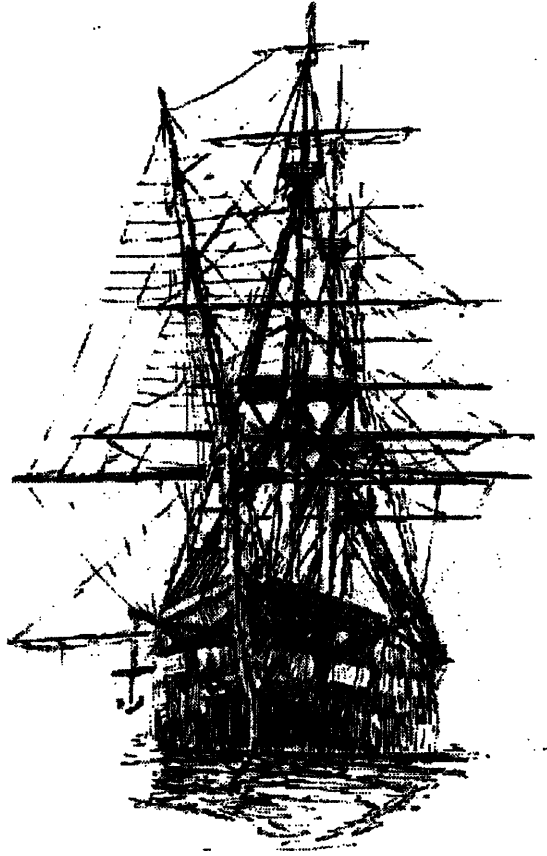


Figure 15 The USS Constitution (courtesy [35])

Another task aboard the Constitution cannot be performed by humans at all. Inside the ship, near the bottom of the hull, ballast planking supports 190 tons of lead and rock used to prevent the ship from rolling. This space is shown in figure 16. The underside of this ballast decking is often wet from water drainage, and must be thoroughly checked for rot. However, there exists very little room between the decking and hull; no human could possibly fit into this area to inspect the planking underside.

One potential solution to this problem is to send a small, autonomous robot into the inspection space to check for rotting. If rot is detected, the robot could be instructed to spray the rot-infestation with a borate compound that inhibits rot growth.

A robot designed to accomplish this task would be outfitted with sensors that detect the presence of rot. Two typical sensors used for this purpose are illustrated in figure 17. One

sensor is a “pick-ax”-type probe which prods the wood and records the wood compliance. Overly-compliant wood indicates the presence of rot. The probe technique requires careful control of the environmental interaction force, as well as the location and orientation of the tip. A robot fitted with this probe requires a limb dexterous enough to manipulate it, as well as the necessary motion plan to coordinate this sophisticated task.

Another sensor has been suggested [28] which bombards the wood with small streams of alpha particles, and records the quantity of particles that get reflected back (see figure 17). The quantity of back-scattered particles is directly related to the integrity of the wood. A robot fitted with this sensor would only be required to “point and shoot” the sensor and record the results. The advantage of the back-scatter sensor is that it uses a non-destructive technique and requires less manipulation from the robot. While back-scatter technology is a promising approach, it currently has not been implemented for field environment applications.

Whichever sensor is fitted on the robot, the robot must be capable of locomotion. The robot must be able to autonomously travel from some insertion point near the keel (see figure 16) throughout the entire inspection space. This locomotion must feature robust stability, so that the robot does not fall and damage itself.

Also, since it would be impractical to attach power cables to the robot, the locomotion must be efficient enough to provide thorough coverage of the inspection space within the power limits of an on-board battery pack. For the class of limbed robots small enough to accomplish this task, an on-board battery may offer up to 1.5 hours of autonomous power [28]. Of course, actual results vary with robot configuration and walking gait used.

Kinematic constraints also effect robot configuration and locomotion. The robot must be small enough to inspect the entire ballast area. If locomotion is achieved via limbs, the movement of the limbs must not interfere with the environment. Furthermore, the robot must fit in the extremely small point of insertion (already illustrated in figure 16).

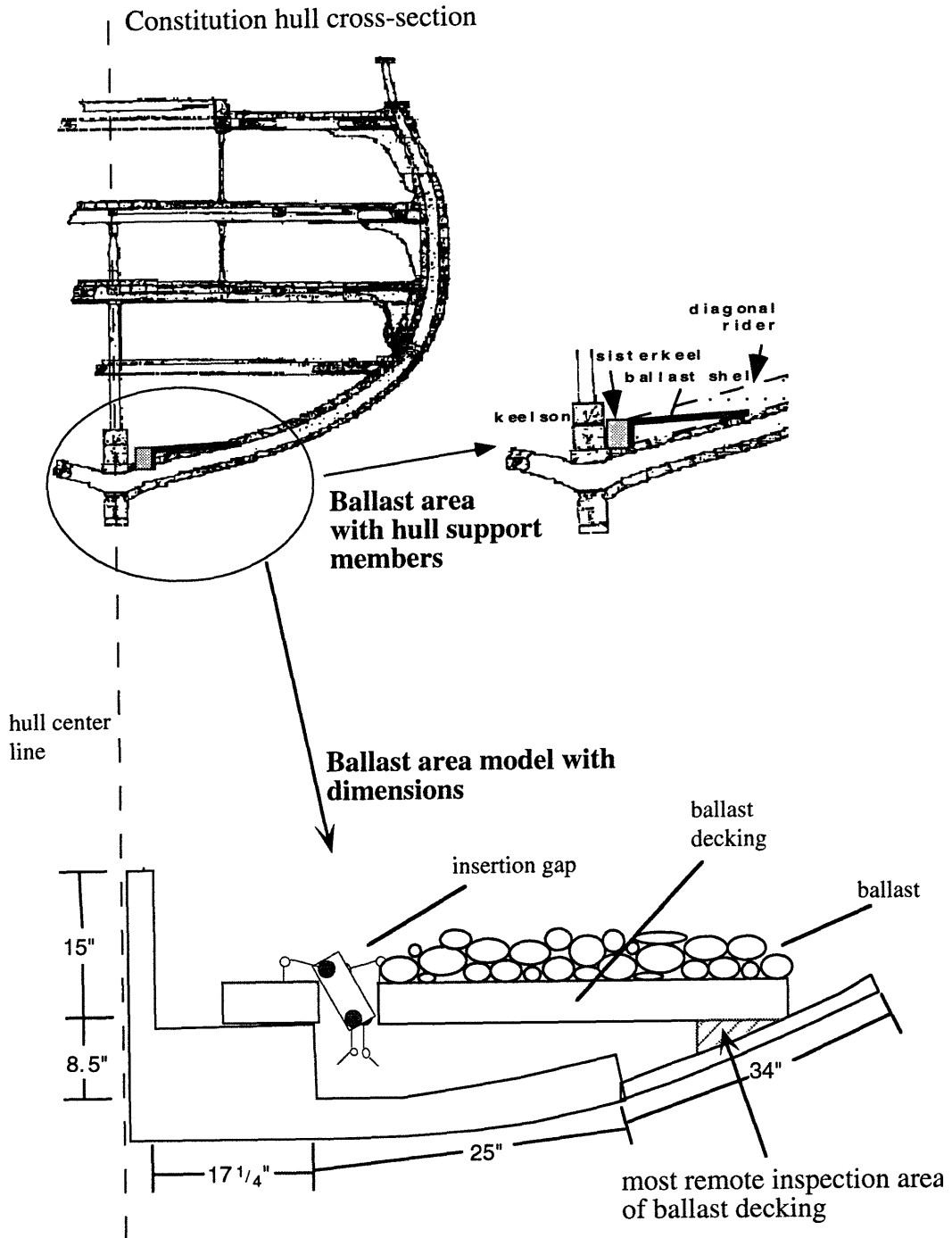


Figure 16 The ballast decking inspection area aboard the Constitution

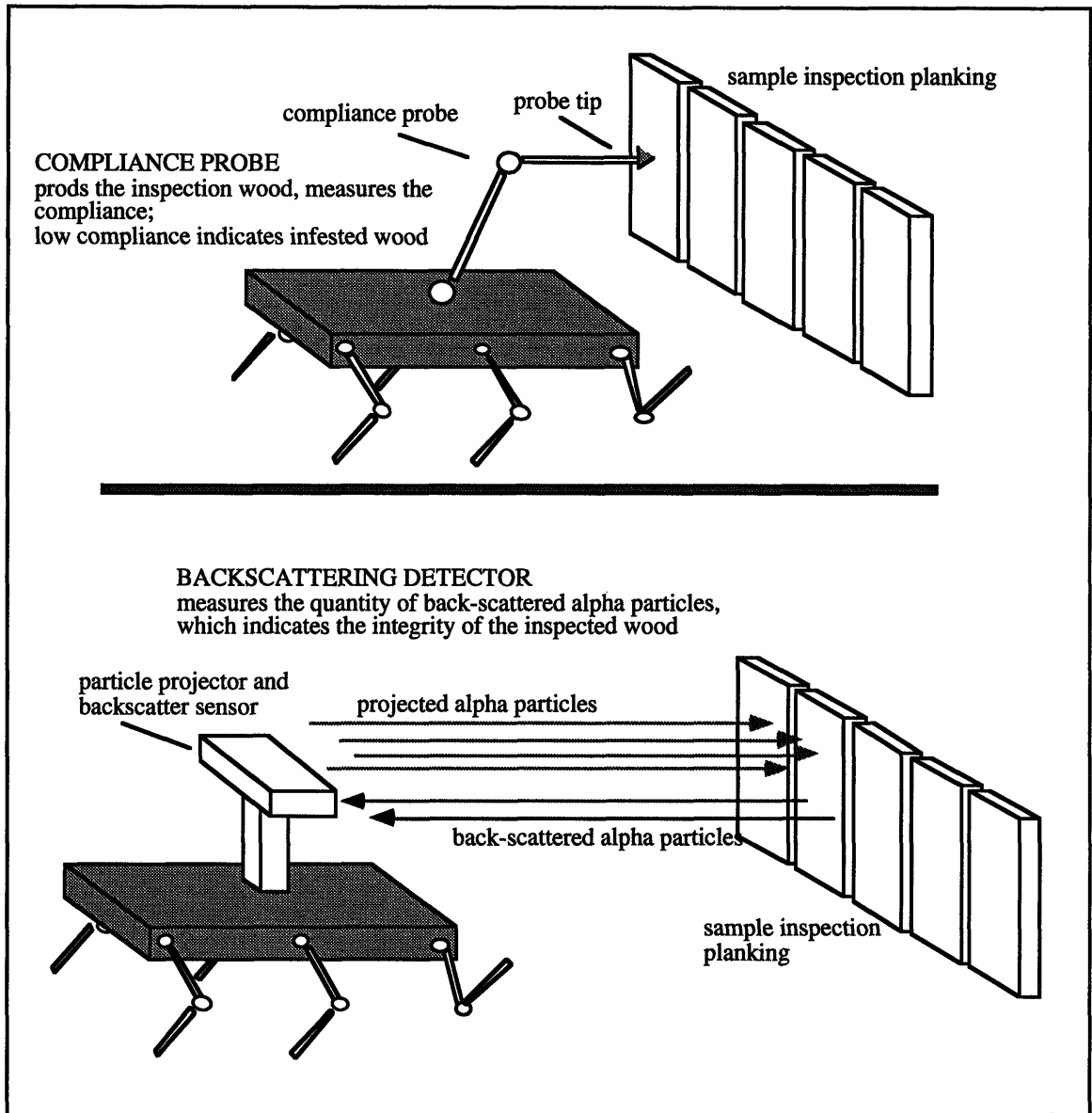


Figure 17 Sensors for the ballast decking inspection task

It should be noted that the proportion of the ballast area dimensions change as one travels along the length of the hull. The ballast area is largest in the center of the ship, and gets smaller traveling fore or aft. The robot configuration and generated motion plan must accommodate this dimensional graduation.

Due to the unstructured nature of the Constitution environment, several other factors affect robot locomotion. The dimensions of the ballast area are only known to a nominal accuracy, based on drawing specifications. During actual application, the robot may

encounter variations in environmental dimensions. Natural wear, cracking, or breaking of the hull planking can effect the contour of the to exposed planking surface. The dimensions shown in figure 16 are expected to be accurate to within plus-or-minus 1". A robot motion plan must be able to accommodate potential environmental variation within this range.

Topological changes to the Constitution ballast area, such as fallen timber, debris, or loose ballast might affect task accomplishment. Approaches have been suggested (see Chapter 3) to generate plans for this class of unstructured environments featuring topological changes. However, the method proposed by this thesis addresses the class of environment which features dimensional variation within a predicted range.

Given these task requirements and constraints, several robot configurations are capable of solving this task (see figure 18). One robot features a center body with a tread on each side for locomotion. An appendage is affixed to the center body, used to manipulate either a probe or back-scattering sensor.

Another robot features a center-body with attached limbs for locomotion and sensor manipulation. The number of limbs could vary from four (one to manipulate the sensor and three being the minimum number required to achieve static stability) to an almost limitless number that does not violate kinematic, power, or other constraints.

For each of these configurations, the hardware components used to assemble the robot could be selected from several sizes. With a fully-functional rapid-deployment modular robotic system, the rapid designer would suggest robot configurations and sizes that were capable of accomplishing the task.

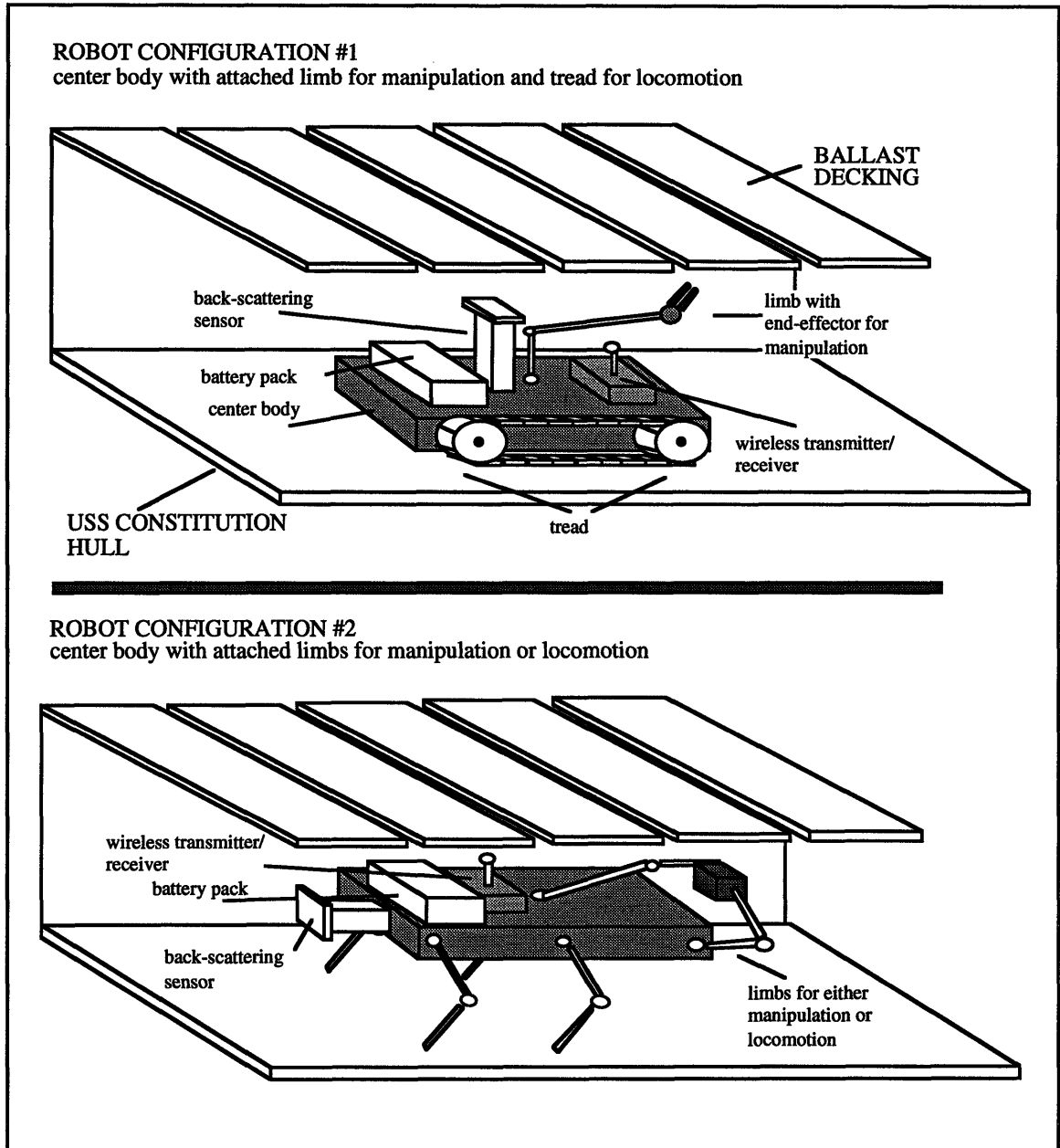


Figure 18 Potential robot configurations for the ballast decking inspection task

4.2 Modeling the Constitution application for rapid plan generation

Two robot configurations were selected to solve the Constitution ballast inspection task. A rapid plan generator was constructed to solve the Constitution application for the selected robots. In order to achieve this, the task was simplified, and simulations models were

made of the robots and task environment. This section describes the simplified environmental model, task description, and robot models.

4.2.1 Environmental model

For the purposes of rapid plan generation, the environmental model is simplified (see figure 19). The environment is assumed not to vary along the dimension into the page (along the hull of the ship), essentially making the problem two-dimensional.

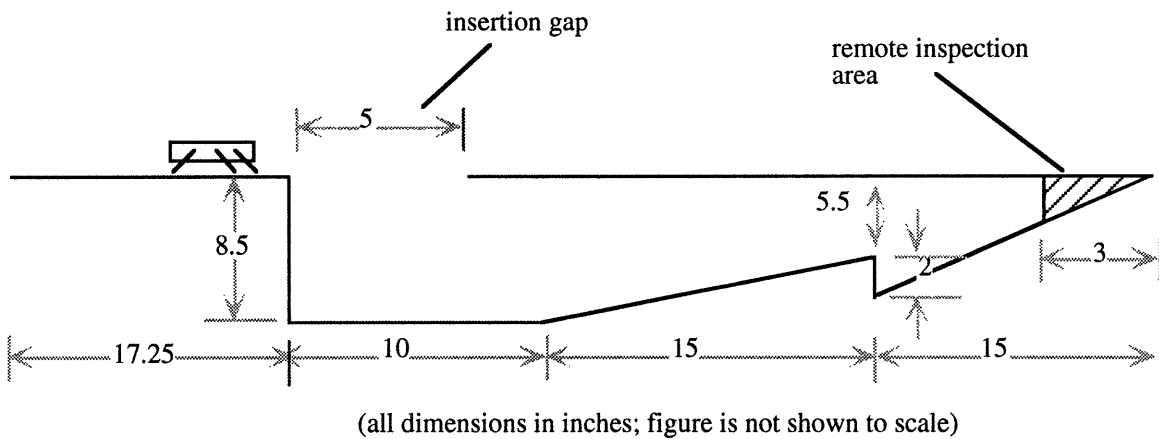


Figure 19 Model of the Ballast Inspection Area

The environment model features zero compliance and an infinite coefficient of friction. Essentially, friction and compliance were “modeled out” of the problem. This is a valid assumption for this task, since compliance and friction play little part in task accomplishment.

These nominal-case environmental dimensions are based upon blueprint drawings. As already mentioned (Chapter 1), unstructured environment dimensions vary considerably by the time the robot system is deployed. Actual measurements indicate that any of these parameters may vary by up to an inch.

4.2.3 Robot model

Many robots could have possibly solved this task. In a fully-functional rapid-deployment mobile robot system, the rapid designer would suggest a robot configuration each time through the design cycle (see the overview of the system design cycle in Chapter 2). For this sample application, four robots were selected which satisfy the task criteria described in the last section.

Two of the selected robots are illustrated in figure 20. These two robots share the same basic configuration but have vary with respect to the length of the center body. Both robots consist of a center body and three limbs evenly spaces along both sides of the longitudinal axis. One of the robots has a center body length of 5", the other a length of 4". The other robots (not shown) share the configuration of these two robots, but have center body lengths of 2.5" and 7".

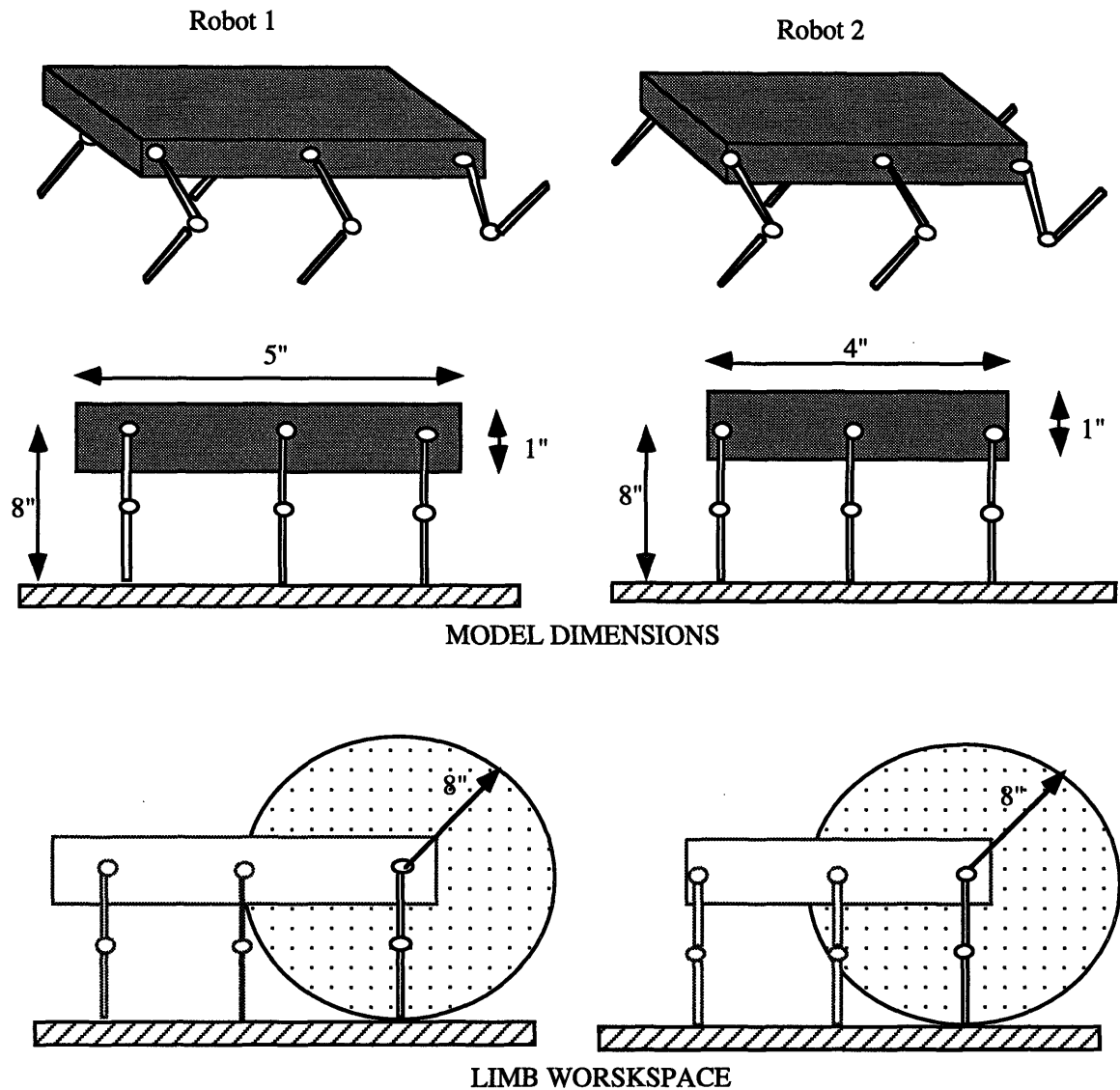


Figure 20 Models of two robots used in the example problem

The robots' length was varied because it is expected that the length of the robot will critically effect the robot success. Figure 19 illustrates that the robot must insert itself into

the ballast area through a nominal 5” gap. The longer the length of the robot body, the more difficult this task becomes. A rapid plan generator should generate successful motion plans for both robot configurations.

For each robot, the following parameters are used to describe the robot position and orientation in inertial space :

- the center of mass’s x-axis and y-axis location.
- the center body’s z-axis rotation with respect to horizontal
- each end-effector’s x-axis and y-axis location

These parameters, referred to as “state variables” for the system, are shown in figure 21.

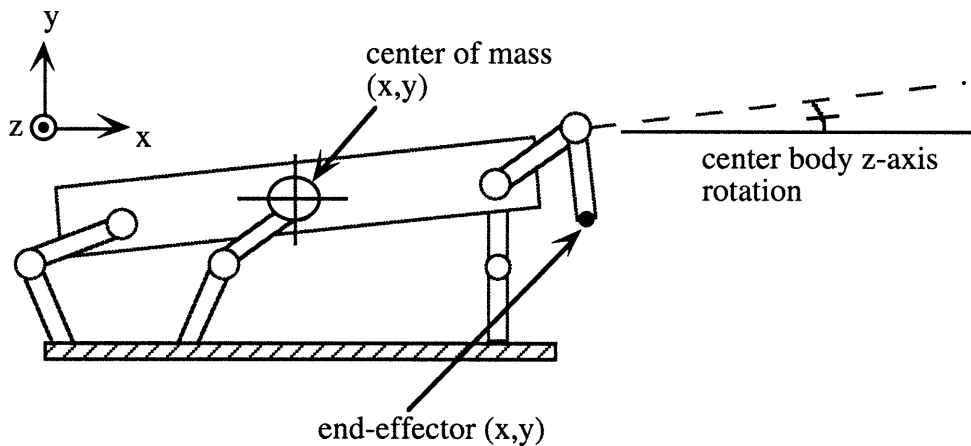


Figure 21 Parameters which describe a robot’s position and orientation

The center of mass is defined to be the exact center of the robot center body. The location and orientation of the robot body is thus calculated based upon the position and orientation of the robot center of mass, and the known length, width, and height of the robot.

It is assumed that the kinematic workspace of each limb is constrained only by the limb length (as illustrated in figure 20). It is assumed that each robot limb has sufficient degrees of freedom to allow end-effector movement anywhere in this workspace. In figure 20, the robot is pictured as having two rotational joints on each limb (shoulder and elbow). The only parameter recorded for a limb is the end-effector position. The actual configuration of joints to achieve the end-effector position is not calculated (that is, inverse kinematics were ignored).

Many assumptions are made about the selected robots in order to simplify the robot models. These assumption simplified the simulation used to evaluate the robot

performance. Using a computationally simple simulation reduces genetic algorithm cycle time. The rapid plan generator is expected to support a more complex simulation model if desired (see next chapter).

The first simplification of the robot model is that the robots may be considered two-dimensional. As will be shown in the next section, the environment model does not change along the axis parallel with the robot width. Thus, the robot width has no bearing on the problem. One can assume the robot to have any arbitrary width.

Also, a robot limb is assumed to have sufficient actuator power to apply any desired environmental interaction force without saturating any actuators. The limbs' main purpose for this example will be to provide locomotion and support the center body. The desired environmental interaction forces will be to support the body weight for these task. As long as the robot body weight is sufficiently low, this model assumption is not unrealistic.

The end-effectors are assumed to provide infinite friction with the environment. This is a valid assumption, since realistic values of slip forces will not be achieved in accomplishing this task. Essentially, environmental friction is modeled out of this problem.

Finally, motion is assumed to be of such low speed that no dynamic effects occur.

4.2.2 The task description

For the rapid plan generation application, the task is also simplified. A solution will be considered successful if it travels from the insertion point to the most remote area of inspection (shown in figure 19). Manipulation of rot sensors, while compatible with this approach, is not considered part of the task.

The task is further simplified in that the robot is only required to inspect a ballast area at only one location along the hull. As already discussed, the ballast area dimensions vary along the length of the hull. An actual implementation must generate motion plans to accommodate the entire range of ballast area dimensions.

Robot locomotion is subject to the constraint of static stability. Static stability is violated when the projection of the robot center of mass (located in the center of the robot body) in the direction of gravity falls outside of the polygon defined by end-effector contact with the environment. The constraint of static stability is illustrated in figure 20. If at any time during robot locomotion static stability is violated, the robot has considered fallen, and the task fails.

A robot may not execute an instruction which yields environmental interference. If environmental interference occurs (as shown in figure 21), the robot motion stops.

Power limitations are modeled as a limit on the number of instructions the robot is allowed to execute. After 1000 instruction executions, the robot is considered out of energy, and may execute no more instructions.

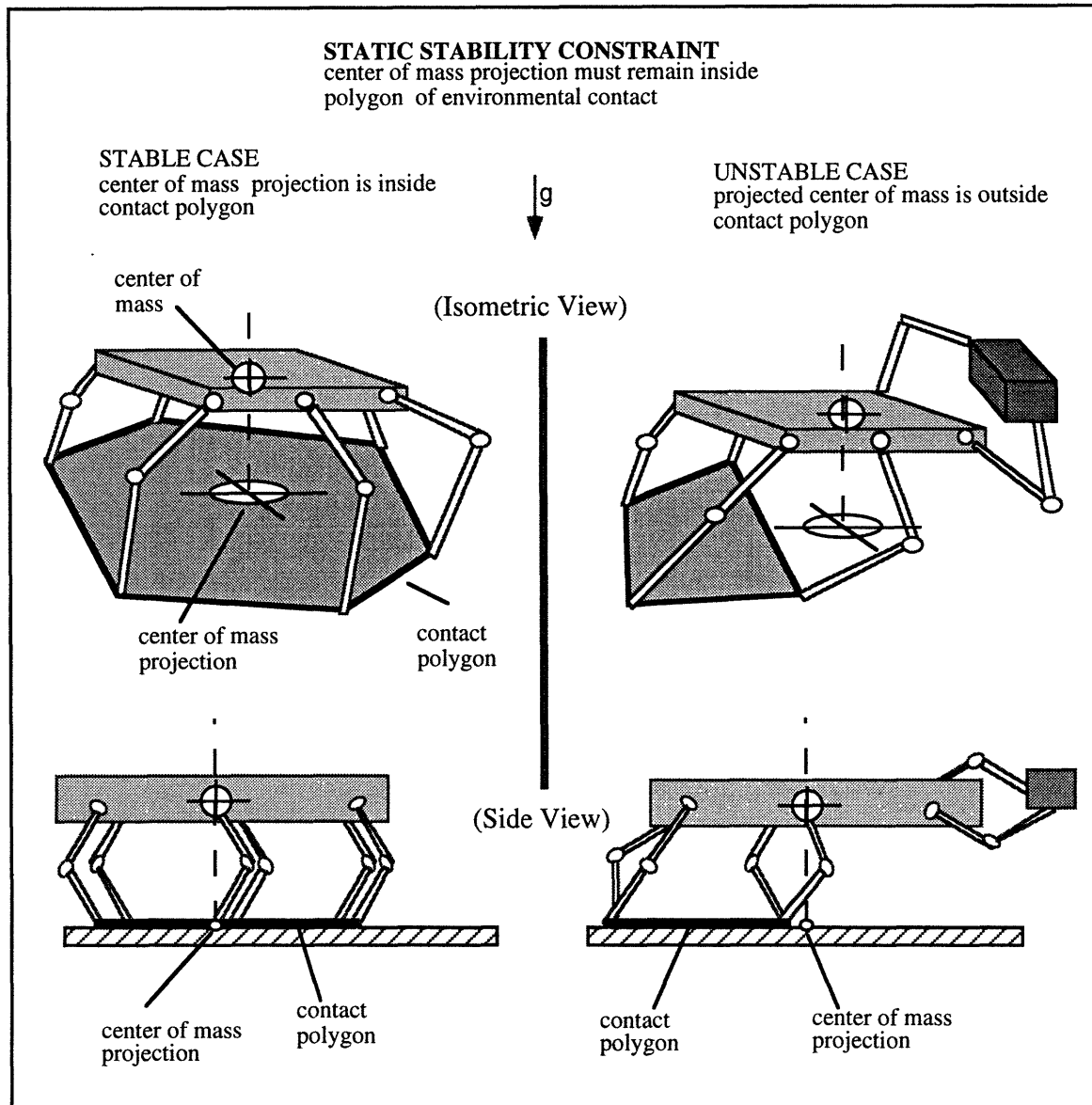


Figure 22 Static Stability Constraints

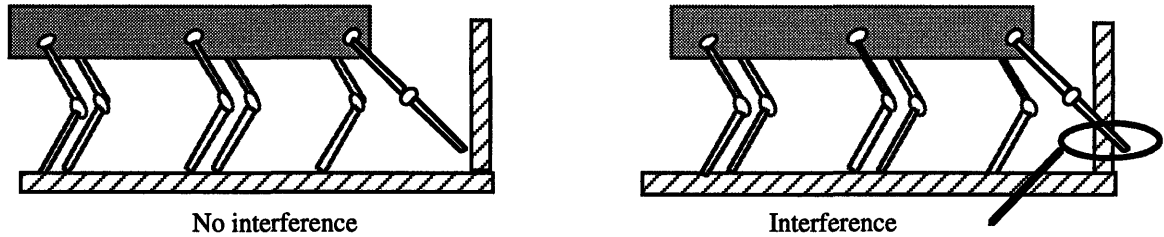


Figure 23 Environmental Interference Constraints

4.3 Creating the rapid plan generator for the Constitution task

A rapid motion plan generator was created for the Constitution ballast inspection task and the selected robot configurations. In order to implement this generator, an inventory of action modules was created and a fitness function was defined. This section describes the creation of this components.

4.3.1 Creating an inventory of action modules

As described in the last chapter, a fully-functional rapid plan generator includes a formal methodology for selecting useful modules based on the application characteristics. For this example, a set of action modules were selected based upon an ad-hoc assessment of predicted task requirements.

The task for this example problem is to achieve locomotion from the point of insertion to the most remote section of the ballast area. In order to achieve this, the robot must be capable of locomotion which does not violate static stability or power constraints, and it must be capable of squeezing through the insertion point gap.

Six modules were used to command body center of mass movement. One module each was used to command the center of mass forward, backwards, up, down, rotate counter-clockwise, and to rotate clockwise. Figure 24 illustrates the effect of these modules on body position for a given initial position configuration.

Each of these body control modules instructed the center of mass to move in finite increments. For example, the “Body Forward” module commands the center of mass forward 1 inch, and the “Body Backwards” commands the center of mass backwards 1 inch. The magnitude of these commanded movements was arbitrary within the kinematic constraints described.

For each body command, the end-effectors remain stationary. As long as the commanded movements do not violate limb length constraints, it is assumed that some joint configuration achieves the desired body command. For a redundant system such as the six-limbed robot in this example, this assumption is valid. For actual application, other criteria would be needed to dictate joint configurations for this under-constrained system.

As already discussed, the task requires that the robot achieve locomotion. Locomotion is comprised of repetitive step-taking actions. To facilitate plan generation, a “step” action module was created. This “higher-level” action module commands each limb to take a step one inch forward (see figure 25). Each module contains the following string of low-level actuator instructions:

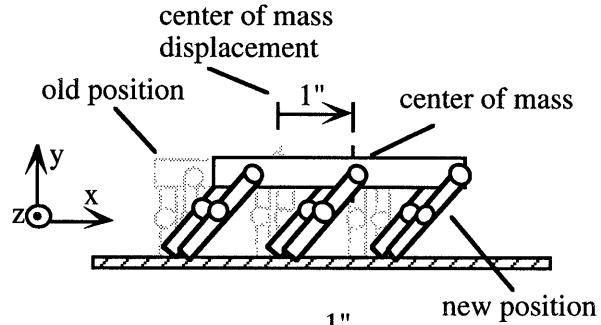
- lift end-effector 1 inch
- move end-effector forward one inch
- drop end-effector until it contacts the environment (or kinematic limit is reached)

The distance that the leg was to move down was purposely left sensor-dependent so that the command would always work within limits of varying environmental contour. As discussed, the Constitution environment may vary upon actual application.

It should be noted that an alternative approach to selecting the leg action modules would have constructed one module for each of the instructions “lift end-effector”, “move end-effector forward”, and “drop end-effector.” However, it was clear that these instructions would be strung together many times within the script, so one action module was created which contained the entire sequence. It will be shown in the next chapter that creating these higher-level action modules which solves task-specific problems quickens solution generation time.

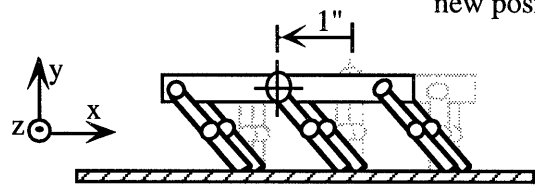
"Body Forward"

move center of mass in positive x-direction 1 inch



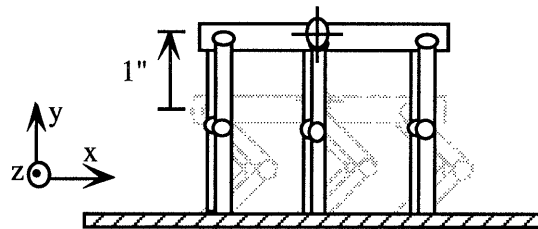
"Body Back"

move center of mass in negative x-direction 1 inch



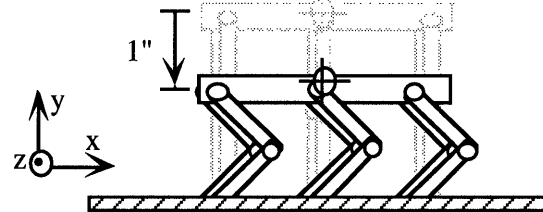
"Body Up"

move center of mass in positive y-direction 1 inch



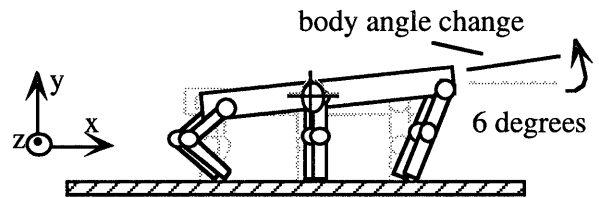
"Body Down"

move center of mass in negative y-direction 1 inch



"Rotate CCW"

rotate center body about center of mass 6 degrees in positive z-direction (counter-clockwise)



"Rotate CW"

rotate center body about center of mass 6 degrees in negative z-direction (clockwise)

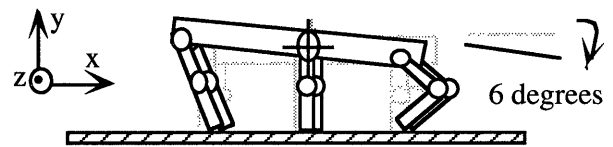
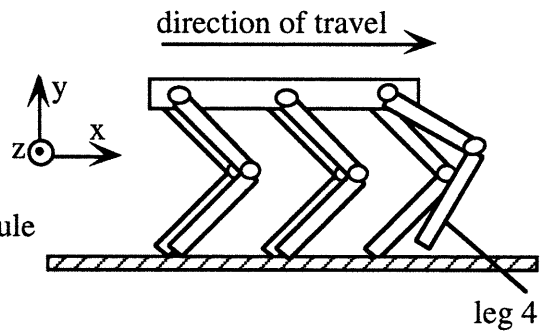


Figure 24 Action modules for controlling body movement

The "Leg 4 Forward" action module commands leg 4 (shown) to take a one inch step forward.

In the rapid plan generator, a similar module was used for each leg.



"Leg 4 Forward"

1. move leg 4 endpoint in positive y-direction 1 inch
2. move leg 4 endpoint in positive x-direction 1 inch
3. move leg 4 endpoint in negative y-direction until environmental contact made by endpoint

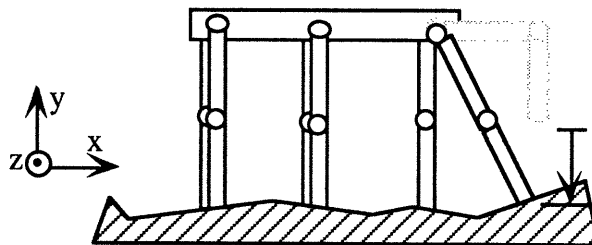
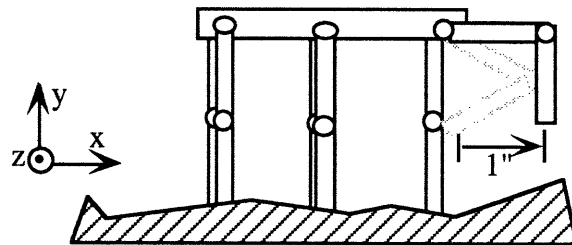
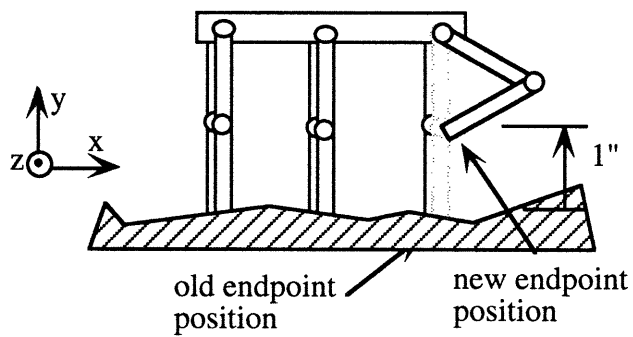


Figure 25 Action module for controlling limb movement

4.3.2 Simulating a script's performance

The genetic algorithms approach requires that each offspring script be assigned a fitness score, so that parent scripts can be selected for reproduction. The script's fitness is a measure of how well a robot performs a specific task while executing that script. In order to measure this fitness, the robot performance is simulated in the task environment, and a fitness function assigns a score based on that performance (deriving a fitness function for this task is described in the next section).

Thus, a simulator was written to evaluate the performance of either of the selected robot configurations within the ballast environment executing a generated script. For a specific robot configuration and script, the simulator keeps track of the robot state variables (as described in section 4.2.3.)

The action modules contain instructions which command changes in the state variables only. For example, a "Body Forward" module causes the center of mass of the robot system to be move in the positive x-direction one inch. Since the robot center of mass is modeled as the center of the robot body, the rest of the body translates with the center of mass. The entire body geometry is easily calculable based on center body position and orientation, and robot length and width.

For a script to be evaluated, each action module is executed sequentially. The robot state variables are updated to reflect the robot's execution of this command. Then, the constraints of kinematic limitations, environmental interference, and static stability are all checked. If the constraints are violated then the simulation of the script ends, and a fitness score is assigned. If no constraints are violated, the simulator proceeds to the next action modules in the script. A flowchart of a simulation cycle is shown in figure 26.

4.3.3 Defining the fitness function

In order to implement the rapid plan generator, a fitness function was created to evaluate the success of each generated script. The genetic algorithm then picks the best scripts to pass along their genetic information to a new generation.

For this task, success is defined as getting to the inspection site already denoted. However, the fitness function needs to determine the relative success of every solution, whether the task is accomplished or not. Some criteria must be created to achieve this.

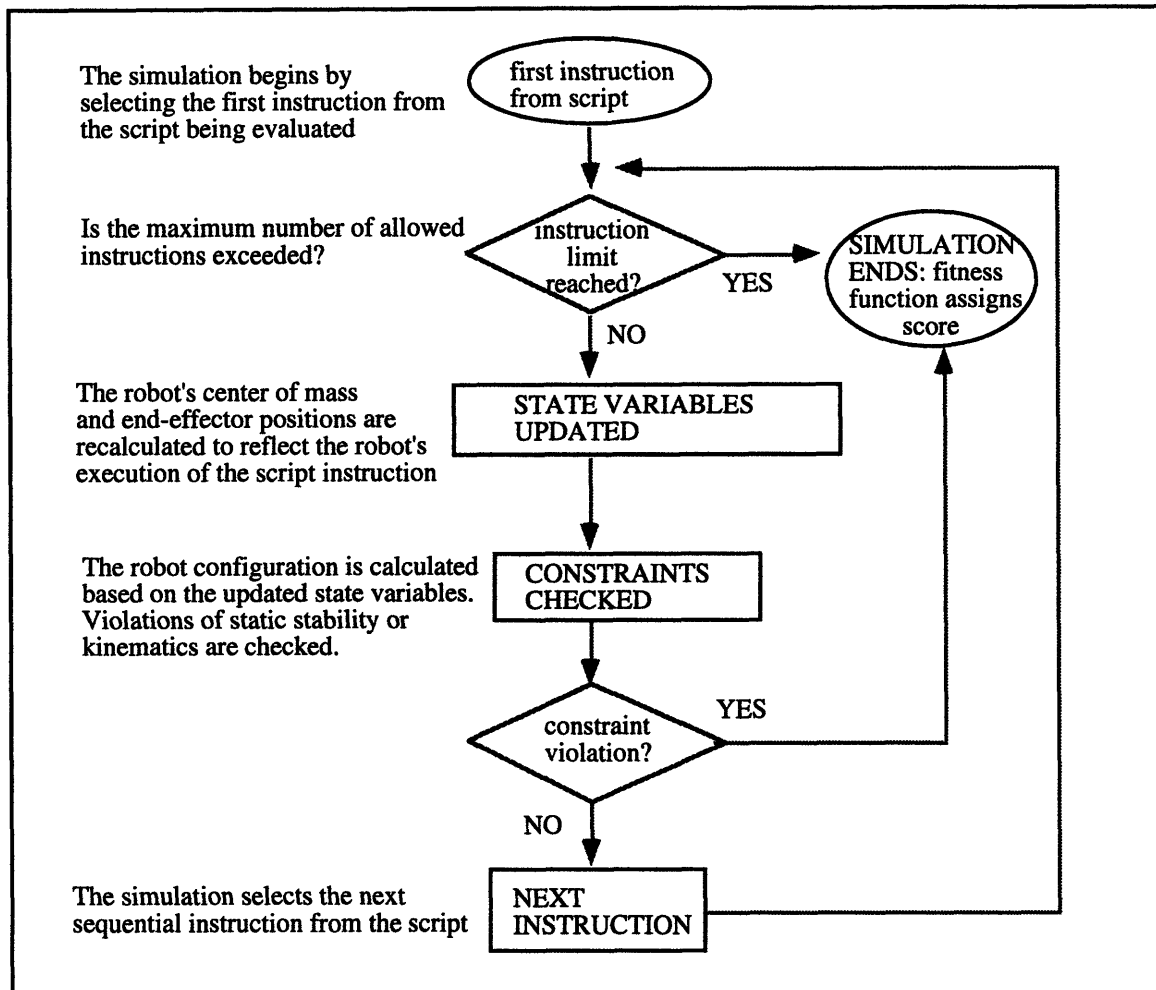


Figure 26 The simulation cycle for evaluating script performance

For this example problem, one criteria of relative success is the distance the robot travels towards the destination site. Using this measure, the fitness score is equivalent to the distance traveled as denoted in figure 27. The distance is measured from the point of insertion until the robot stops moving because of either task accomplishment, violation of static stability, or the limit of instructions is reached.

It is suspected that the scripts which command the robot to travel farther contain more useful “genes” for accomplishing the task. By using this distance criteria, these scripts are selected to pass on their genetic information.

In this fitness function, distance traveled was the only criteria of success. Other criteria could have been included, such as the amount of power conserved or speed of task accomplishment. These criteria were not deemed important for this example problem. In a

fully-functional rapid plan generator, relevant fitness function criteria would be selected using experience-based rules and the application characteristics.

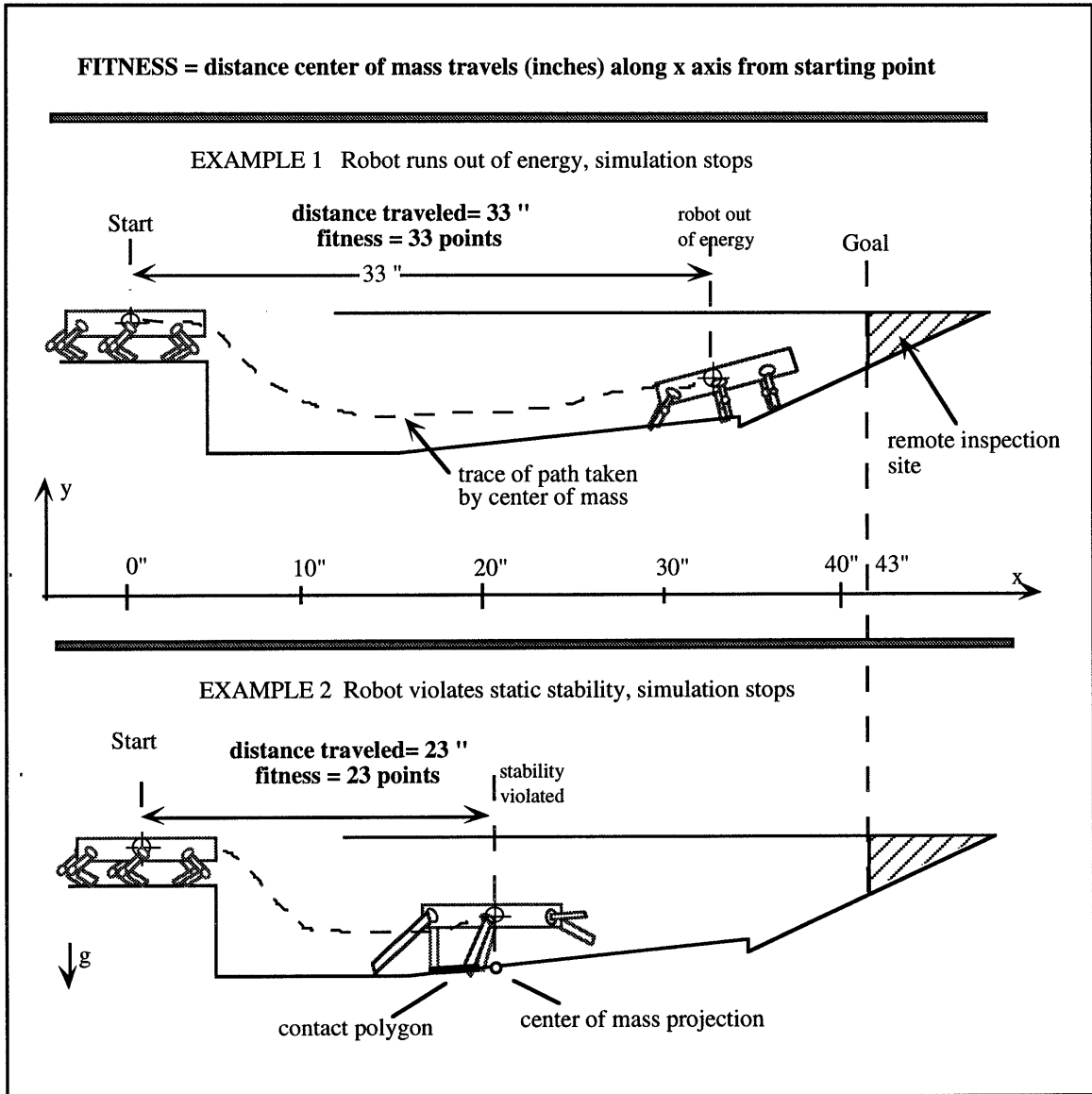


Figure 27 Robot success as defined by the fitness function.

4.3.4 Acknowledgment: authorship of the genetic algorithm software

Part of the software used for this rapid plan generator was written by Singleton [17]. Singleton's software executed the genetic algorithm selection and reproduction mechanisms for a user-provided set of genes and fitness function.

Using Singleton's genetic algorithm framework, custom software was written to represent the action module genes.

To support the evaluation phase of the genetic algorithm, a simulation of the Constitution problem was written. This simulation incorporated the robot and environmental models described in this chapter.

A customization was made to the fitness function evaluation, allowing a solution score to be based on multiple training samples. This customization allows more robust training, and is further described in the next chapter.

Chapter 5 Results from the Constitution application

The previous chapter describes an example application for a rapidly-deployed robotic system to inspect the ballast decking aboard the USS Constitution. The task is defined, and models of the environment and robot configurations are presented. The rapid plan generator for this task is discussed, including the action module inventory and the fitness function.

This chapter discusses the results from this example application. Several results can be used to improve the rapid plan generation technique. Including useful “higher-level” action modules in the inventory reduces solution generation time. Also, plans’ robustness to environmental variation is improved by training them to perform in environments with the maximum predicted variations. Finally, one inventory of action modules may generate successful solutions for a variety of robot configurations.

5.1 A typical solution

Figure 28 shows a typical generated script generated for the 5” long robot, and an illustration of the robot executing that script. In the illustration, the trace of the robot center of mass is denoted by the dotted line. The robot configurations shown reflect “snapshots” of the robot position at several places along the path. (Note that the robot configurations shown do not represent the updated positions for each instruction execution.)

For the purposes of this application, when the robot reaches the end of the script, it continues executing the action modules at the beginning of the script. It stops executing action modules after reaching the goal, failing (violating static stability, kinematic limits, or

environmental interference), or reaching the 1000 module-execution limit. This approach exploits the repetitive nature of locomotion through the ballast area in order to generate smaller scripts. Generating smaller scripts reduced generation time, allowing for more experiments to be run. The generation times for these scripts suggest that generating complete scripts would have been possible within rapid deployment system time constraints (days-weeks).

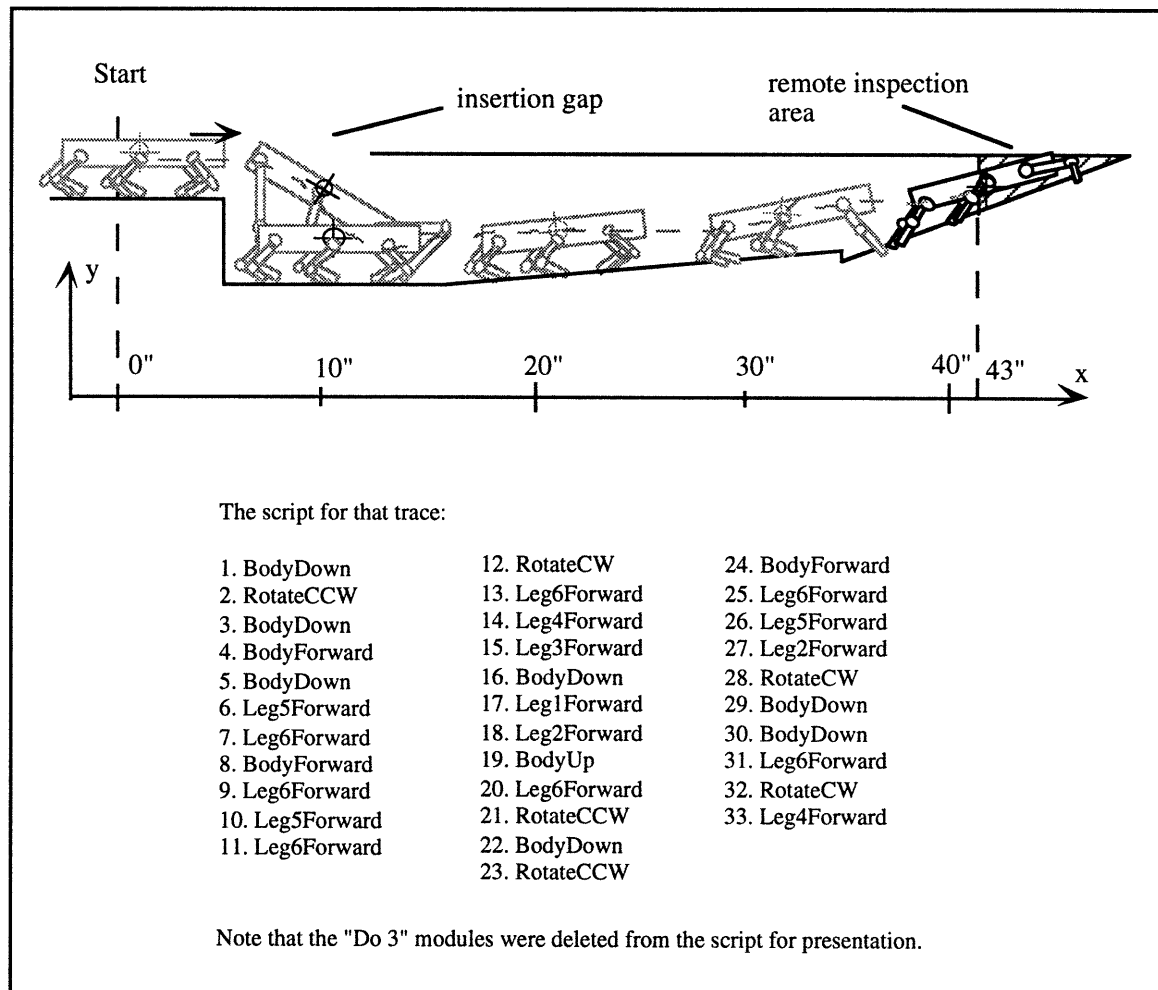


Figure 28 An example script and illustration of the robot executing the script

This script in figure 28 contains 33 action modules and is executed eight times in order to perform the task. The total number of actions modules executed is 33 instructions x 8 repetitions = 264 executed modules. For each script, the number of repetitions necessary to complete the task depends upon script length and how much a single repetition

contributes to performing the task. An average script length is 35 action modules, and is repeated 10 times.

This particular script yielded task success; the robot traveled 43" from the starting location all the way to the remote inspection site. The fitness score for this script is 43 points, reflecting the 43 inches traveled. The time required to generate this solution on a SparClassic workstation was 3 minutes. The genetic algorithm went through 345 cycles in order to generate this script. For this example problem, average script scores were 43 points, requiring 3 minutes and 375 genetic algorithm cycles to generate.

Figure 28 shows only snapshots of the robot position during task performance. The figure does not show how each action module effects robot position.

Figure 29 shows how robot position is effected after execution of 9 consecutive action modules. The action modules were extracted from a generated script. The overall effect of the 9 modules is locomotion of the entire robot system forward one inch. The example script consists of 3 "Body Forward" action modules and 6 "Leg X Forward" modules. Recall that the "Body Forward" action module commands the center of mass to move forward in the x-direction, and the "Leg X Forward" module commands Leg X to take a step forward.

Figure 30 illustrates another sequence of 2 action modules extracted from a generated script. The two modules are "Body Forward." After executing the modules, it leans forward too far; the projected center of mass falls outside of the contact polygon. Static stability is violated, and the robot fails.

Note that the presence of two consecutive Body Forward modules is not sufficient condition for the robot to fail. In the example script in figure 29, two consecutive Body Forward modules are executed without robot failure. The reason the robot falls in this case is that it is already leaning far ahead when reaching these two modules in the script.

Thus, it should not be concluded that a module or string of modules always causes robot success or failure. The presence of the string in conjunction with other specific conditions (in this case, the robot leaning over too far already) *may* in fact cause robot failure.

-
-
-
- ① Body Forward
- ② Body Forward
- ③ Leg 1 Forward
- ④ Leg 2 Forward
- ⑤ Leg 3 Forward
- ⑥ Leg 4 Forward
- ⑦ Leg 5 Forward
- ⑧ Leg 6 Forward
- ⑨ Body Forward
-
-
-

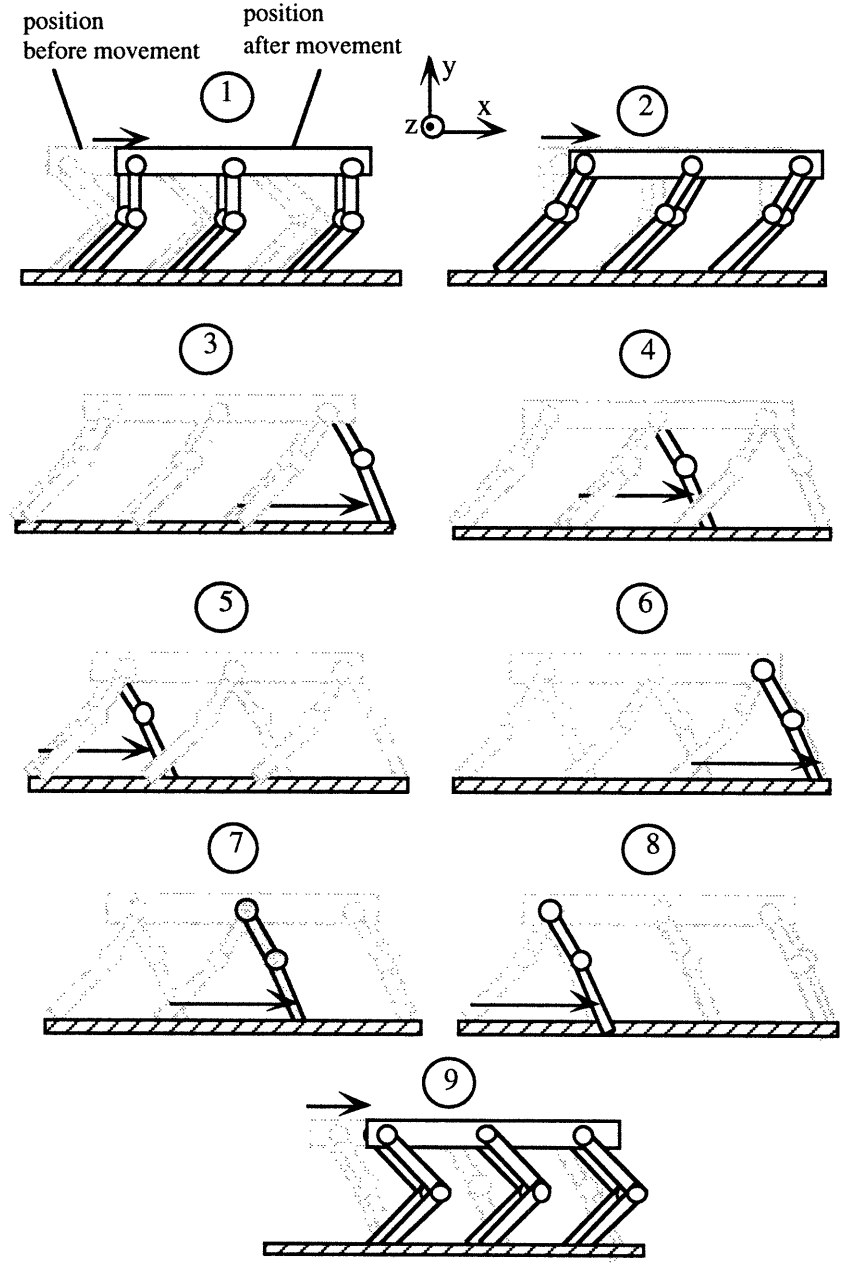


Figure 29 Illustration of robot walking using script

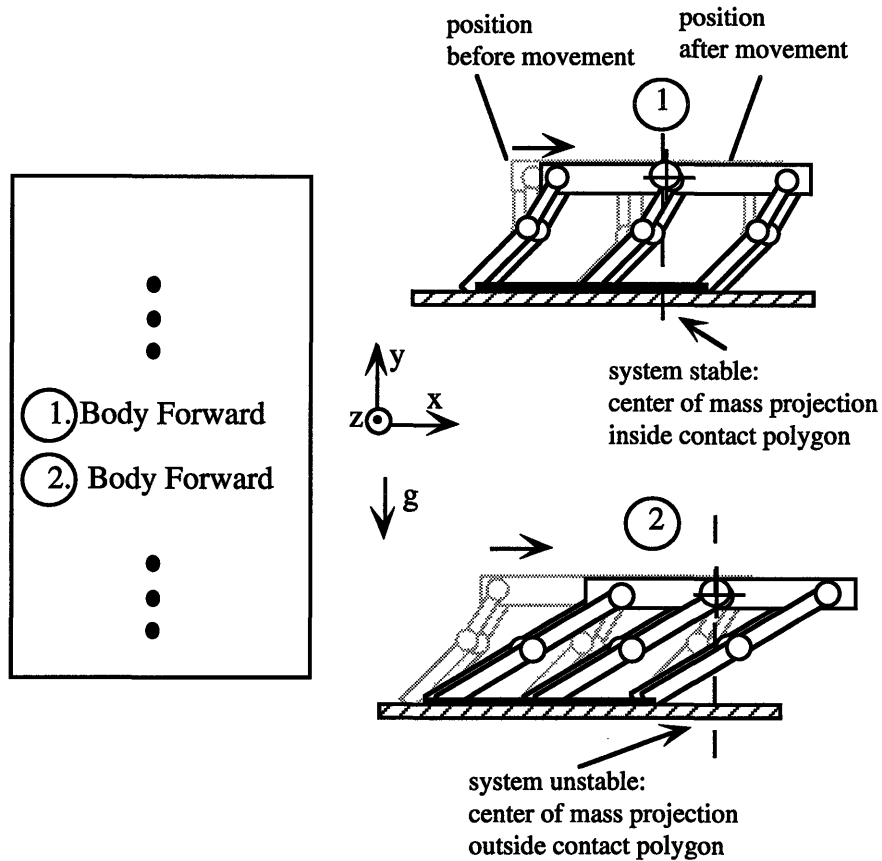


Figure 30 Example of a script causing robot failure

5.2 Improving robustness through training on worst-case problems

The previous scripts were generated for the robot to perform on the nominal-case environmental dimensions. As already discussed (Chapters 1 and 4), some of these dimensions may in fact be different when the robot is deployed.

The environmental model dimensions may be expected to vary by up to 1". In order to test a generated plan's robustness to environmental variation, 6 new environmental models were made. Each of these models features a maximum dimensional change of 1". A script is expected to successfully perform the task on all 6 worst-case models. The robot should also work for a dimensional change within the limits of nominal case and maximum case.

Figure 31 illustrates the environmental models with the 6 dimensions changes. The changes are:

1. Sister keel narrowed (side edge moved in negative x-direction) 1”
2. Sister keel widened (side edge moved in positive x-direction) 1”
3. Hull lowered (moved in negative y-direction) 1”
4. Hull raised (moved in positive y-direction) 1”
5. Sister keel height lowered (top edge moved in negative y-direction) 1”
6. Sister keel height raised (top edge moved in positive y-direction) 1”

Thirty scripts were generated for the 5” robot in the nominal case environment. Each script is the result of 1000 genetic algorithm cycles. The average score for these scripts was 43. Each script was then applied to the 6 maximum-variation environments. The resulting robot scores in these environments is shown in figure 32.

The results in figure 32 show that a plan generated for the nominal-dimension environment may perform poorly when a maximum-variation environment is encountered. The plan only demonstrates acceptable success when applied to environments #4 (hull raised) and #6 (keel height lowered). The other environmental variations degrade the robot performance. One of the variations which degraded performance most was #2 (keel width widened). Note that the extra keel width degrades performance because the insertion gap becomes much narrower.

With unstructured environment applications, the scripts must be successful despite these variations. The following method is proposed to accomplish this.

The method is to train the genetic algorithm to succeed on the maximum-variation cases rather than on the nominal cases. To implement this, the fitness function is changed to evaluate the robot’s success on how well it does on the worst-case.

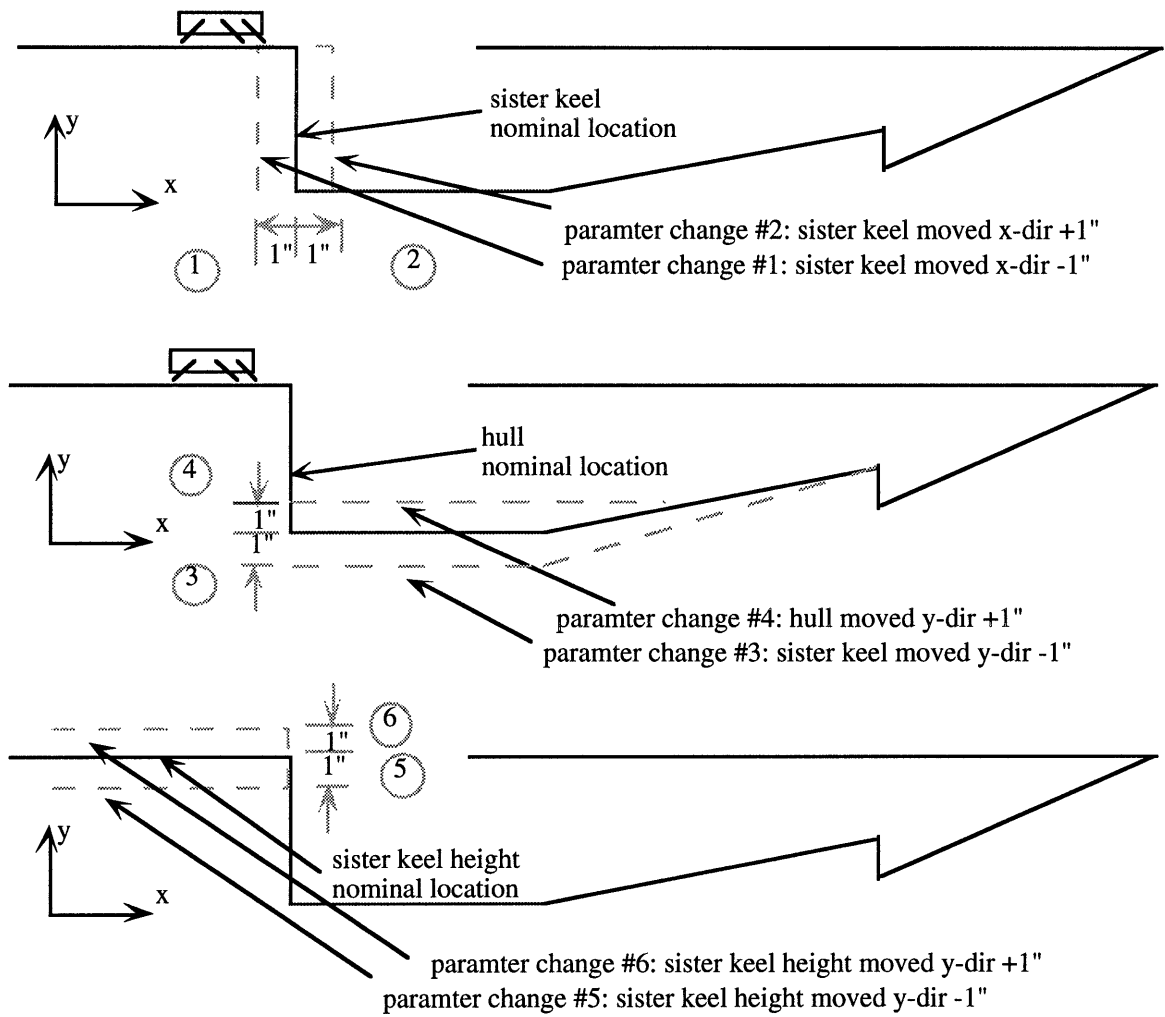


Figure 31 Ballast area parameters' worst case variation

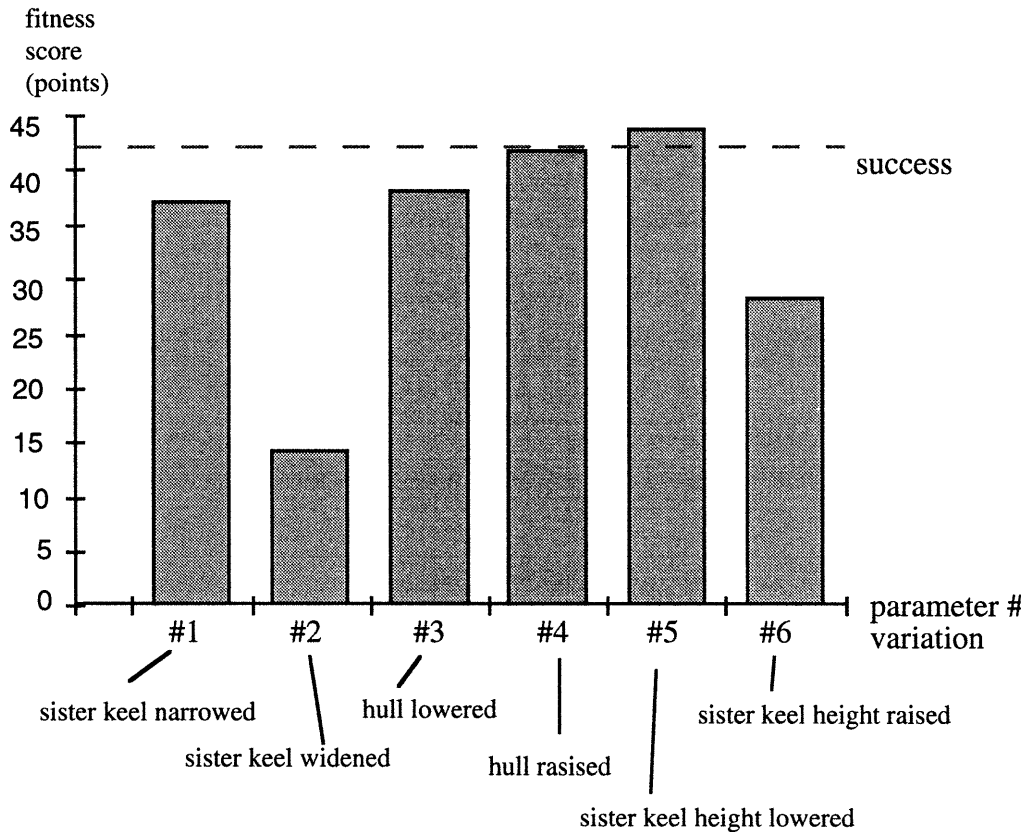


Figure 32 Script performance when applied to cases with environmental variation

To illustrate this method, robot plans were generated to perform on maximum-case variation of parameter #2 (keel width widened).

After a script was generated, its robustness was evaluated by applying it to the nominal case environment and 3 new environments. In these 3 new environments, parameter #2 was varied between nominal case and maximum variation. Figure 33 illustrates these test environments. A robust script performs successfully on all of these cases.

As the results in figure 34 show, robustness is improved through this maximum-variation training. When a script is trained on a maximum deviation environment, the script is successfully applied to the other cases with smaller deviations. However, when a script is trained on the minimum deviation (nominal case), it fails when applied to cases with larger deviation.

The results show that training a script to perform on a maximum-variation environmental parameter may endow the script with robustness to environmental variation over the whole range of predicted variation.

For actual applications, robustness training is a two-step process: first, identifying which parameter changes will degrade script success; second, training the script to be robust to those changes. In this problem, the results from figure 32 implied that the keel width was such a critical parameter. Using the method described in this section, scripts were generated to tolerate keel width variation.

Furthermore, for an actual application, the user might train a script to successfully perform despite several simultaneous environmental variations. This might be accomplished several ways. One method is to evaluate a robot's average success for a variety of environments, each featuring one parameter variation. Another is to evaluate a robot's success on one environment which features several parameter variations. Further research should compare these methods.

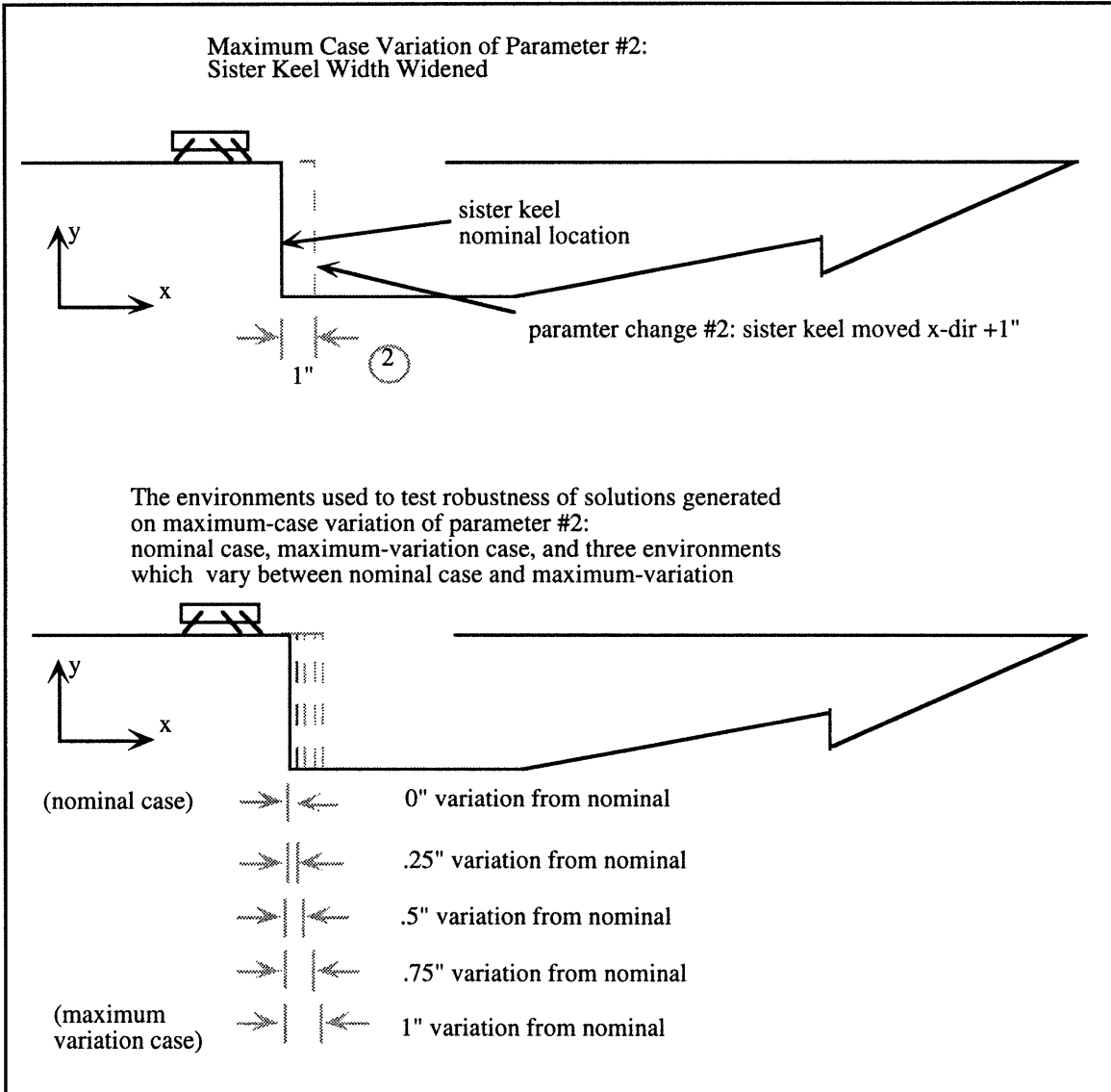


Figure 33 The environments used to evaluate script robustness

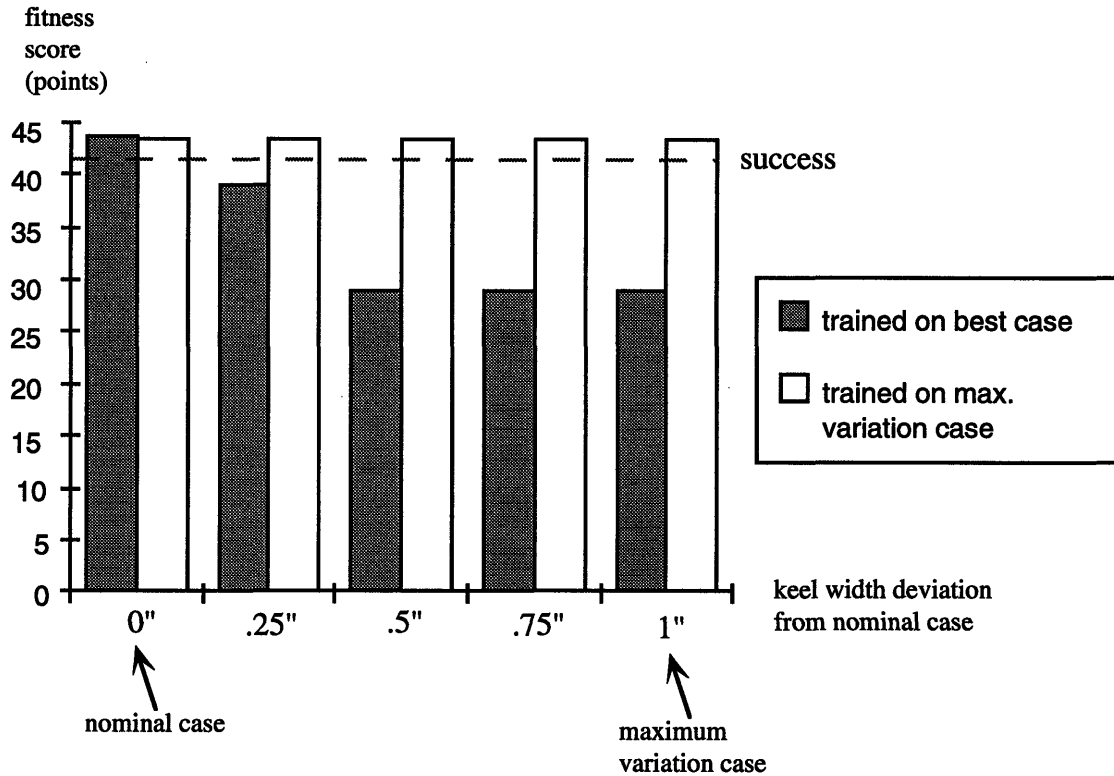


Figure 34 Improved success by training on the worst-case problem

5.3 Improving success by including useful “higher-level” modules in the inventory

Notice in the ballast area problem that the robot must significantly rotate its body clockwise in order to fit through the insertion gap locate 10” from the starting point (see figure 35). With the robot rotated so severely, it has problems squeezing through the rest of the narrowing ballast area. Frequently, it gets stuck before it reaches the inspection area goal. To facilitate reaching the goal, the robot should rotate its body position back to horizontal.

However, to achieve this, the genetic algorithm must generate a motion plan which contains a long string of “Rotate CCW” (rotate body counter-clockwise) action modules that get executed after the robot squeezes through the insertion gap. Since the genetic algorithm assembles action modules into a script at random, it may take many iterations to assemble a particular lengthy string of action modules.

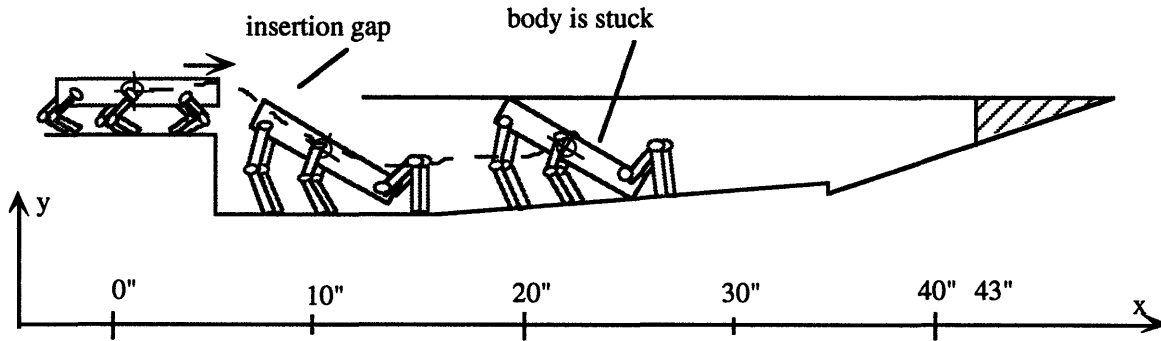


Figure 35 Example of robot "getting stuck"

However, an action module can be introduced which strings together these other action modules, relieving the genetic algorithm from randomly creating the sequence. This module comprised of a sequence of other action modules may be called a "high-level" action module (see Chapter 3).

In this case, the high-level module introduced is called "Body Level." The function of Body Level is illustrated in figure 36. Body Level works by commanding the robot to repeatedly executing a "Rotate CCW" module until the body orientation is measured to be horizontal (parallel with x-axis). Body Level can be incorporated into a script just like any other action module.

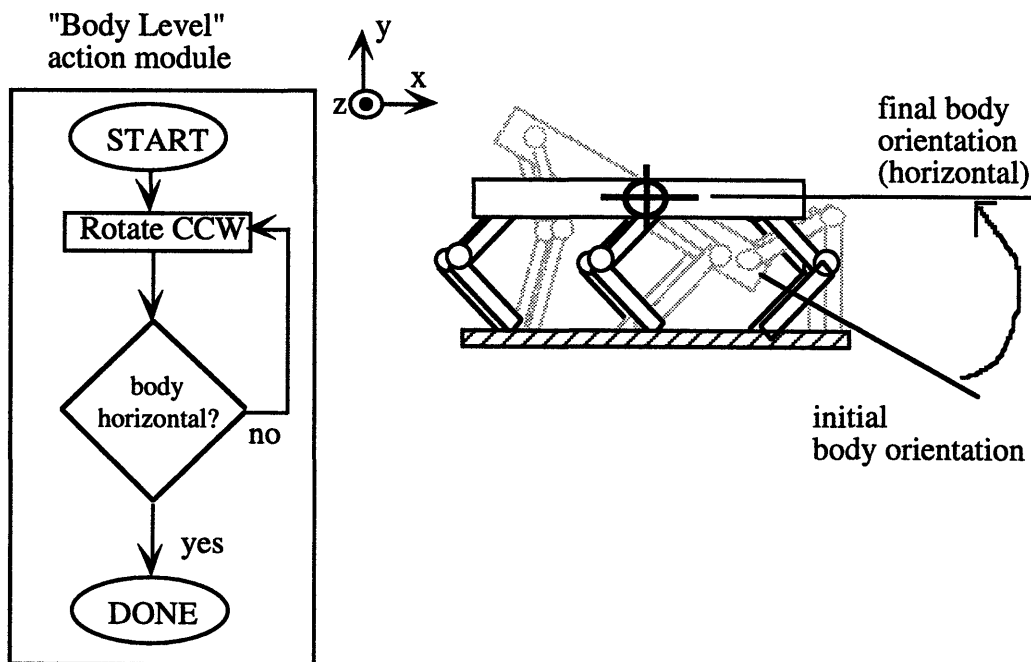


Figure 36 "Body level" module introduced to prevent getting stuck

Since the number of times “Rotate CCW” is executed may vary in each instance of the module, logic and state variable sensing controls this parameter. The process flow of Body Level is also shown in Figure 36, but can be simply stated as: “rotate CCW. If body is not yet horizontal, repeat. If body is horizontal, quit.”

Body Level was included in the action module inventory, and new scripts were generated, trained to succeed on the keel width maximum parameter deviation. The results in Figure 37 show that the successful scripts (score =43) were generated much more quickly using Body Level in the inventory. Successful scripts using Body Level were generated after 325 genetic cycle iterations, or 3 minutes (average of 10 cases). Successful scripts without using Body Level required 1475 genetic cycles, or 14 minutes (average of 10 cases).

Using Body Level generated solutions more than four times faster, but either approach would have generated solutions in a realistic time frame for rapidly deployed systems. However, for more complex applications, the generation time saved using such useful modules may be critical to successful system deployment. Research should develop rules to construct these modules based on task characteristics.

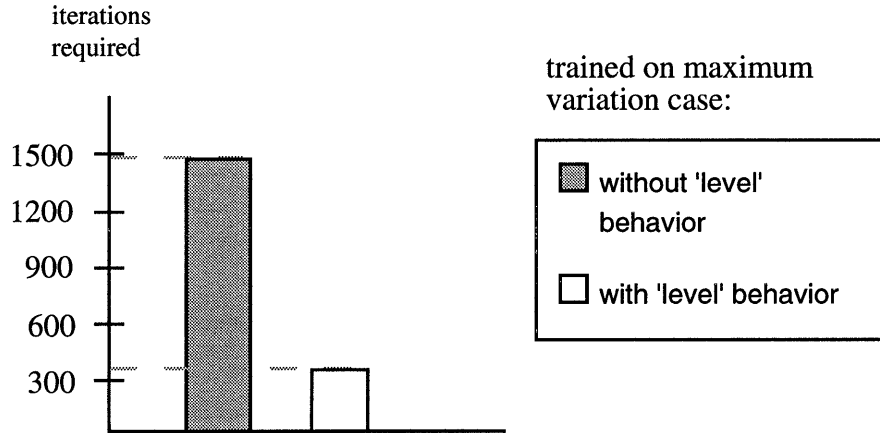


Figure 37 Reducing cycle time by including Body Level action module

5.4 Applicability of an inventory to varying robot configurations

In the last section, scripts were generated for the 5" robot using an inventory which includes Body Level. In this section, the same inventory is used to generate scripts for 2.5", 4", and 7" robots. The 4" and 5" robots are illustrated in Chapter 4.

The scripts were generated for the nominal case environment. It was suspected that the longer length robots would prove less successful because of a more difficult time squeezing through the insertion gap. However, this was not the case. The average script performance generated for each robot configuration was nearly the same, and all demonstrated a successful average (see figure 38).

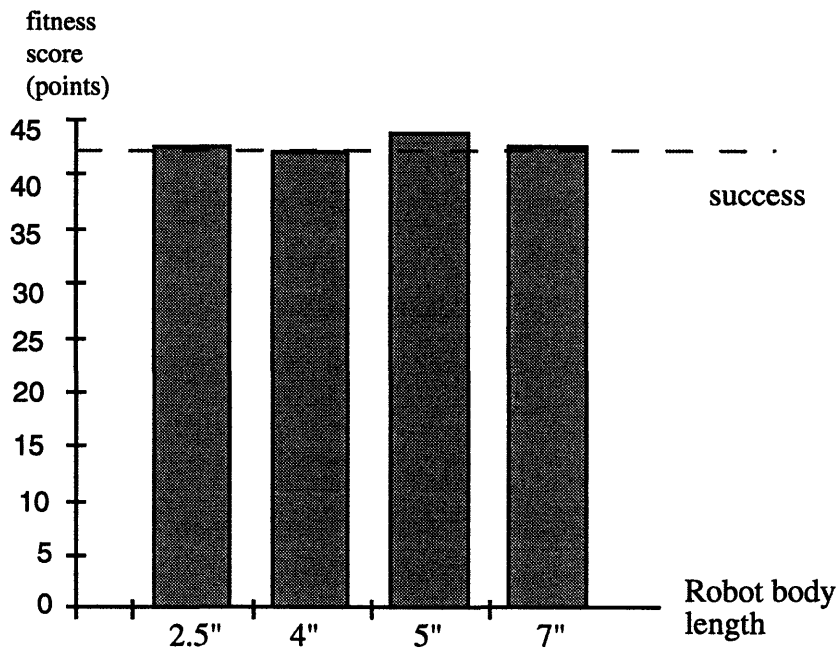


Figure 38 Success for different robot configurations using same action module inventory

The results indicate that one inventory of action modules may generate success for some range of body configurations. However, research should be done in varying other aspects of body configuration, and evaluating the success of the plans. For this problem, drastic variations in leg length or body height may degrade script success using this selected inventory. New action modules may need to be constructed in order to generate scripts for these configurations.

If, in fact, specific robot configurations do require specific action modules to be included in the inventory, rules should be developed for selecting these modules based on robot characteristics.

Chapter 6 Conclusions

6.1 Summary of Results

This results of this research demonstrate that the rapid plan generation technique may be applied to the class of unstructured environments which feature dimensional variability. An original method was discussed which successfully trained a script to succeed over one environmental parameter's range of predicted variability.

Furthermore, results will facilitate the implementation of the technique within the context of a rapid deployment, modular robotic system. Results show that one inventory of action modules may generate a successful script for various robot configurations. Also, a successful script may be generated more rapidly by including action modules which solve task-specific problems.

6.2 Recommendations for future research

There are several focuses for future research. One of the most critical is improving the robustness of these solutions with respect to topological changes in the environment. Usually these topological changes, such as the presence of unanticipated obstacles, or the substantial changes of some other part of the environment, require a plan quite different from the one which solved the training problem. Some approaches to this problem have been suggested, including the generation of subsumptive plans, and including "branches" within a script to conditionally execute "sub-scripts."

A great deal of work needs to be done in further removing the user from the implementation. Currently, each application requires the user to construct an inventory of

action modules, define the fitness function, and develop a simulation for script evaluation. For a truly rapidly-deployed system, some set of rules should map application criteria to a usable set of action modules and fitness function. Further, a general-case simulation environment should be provided.

Some work might be done in using a more complex model of the robot and environment for simulation. Such robot characteristics as dynamics, actuator saturation, probable sensor noise, and realistic surface friction were disregarded in this thesis. The user may wish to incorporate these factors to produce a more result more suitable for actual hardware application.

References

- 1 Anderson, Mary. "Ecological robots." *Technology Review*, Jan. 1992 v95 n1 p22(2).
- 2 Goldsmith, S., "It's a Dirty Job, but Something's Gotta Do It," *Business Week*, Aug. 20, '90, pp. 92-97.
- 3 Stone H. W. and Edmonds G., "Hazbot: A Hazardous Materials Emergency Response Mobile Robot," *Proceedings 1992 IEEE Rob. Automation*, Nice France, Vol. 1, pp. 67-73, 1992.
- 4 Weisman, R., "GRI's Internal Inspection System for Piping Networks," *Proceedings, 40th Conference on Remote Systems Technology*, Vol. 2, 1992, pp 109-15.
- 5 Wehe, D.K., et al., "Intelligent Robotics and Remote Systems for the Nuclear Industry," *Nuclear Engineering and Design*, Vol. 113, No. 2, 1989, pp. 259-267.
- 6 Akizono et al., "Field Test of an Aquatic Walking Robot for Underwater Inspection," *Mechatronic System Eng.*, Vol. 1 No. 3, 233-239, 1990.
- 7 Goldberg,, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.
- 8 Davis, Lawrence. *Handbook of Genetic Algorithms*, Van Nostrand Reinhold. 1991.
- 9 Syswerda, G. "Uniform Crossover in Genetic Algorithms". *In Proceedings of the Third International Conference on Genetic Algorithms*. Morgan-Kaufmann, 1989. pp 2-9.
- 10 Syswerda, G. "A Study of Reproduction in Generational and Steady-State Genetic Algorithms". *In Foundations of Genetic Algorithms*, G.J.E. Rawlins, Ed. Morgan-Kaufmman, San Mateo, CA. 1989.
- 11 Etter, D. M., Hicks, M.J. & Cho, K.H. "Recursive Adaptive Filter Design using an adaptive Genetic Algorithm." *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*. 1982.
- 12 Cooper, Mark G., and Jacques J. Vidal. "Genetic Design of Fuzzy Controllers: The Cart and Jointed Pole Problem." Submitted for the Third IEEE International Conference on Fuzzy Systems; Orlando, Florida, June 1994.
- 13 Hu, Hernt-Tai, Heng-Ming Tai, & Sujeet Sheno. "Fuzzy Controller Design Using Genetic Algorithms and Cell Maps", Presented at the Second International Conference on Fuzzy Theory and Technology, Durham, NC.1993.
- 14 Wieland, A.P. "Evolving Controls for Unstable Systems", In S. Touretzky et. al. (ed.), *Connectionist Models: Proceedings of the 1990 Summer School*. San Mateo, CA: Morgan-Kaufmann Publishers, Inc., 1991. pp. 91-102.

-
- 15 Goldberg, D.E. "System Identification via Genetic Algorithm.", 1981. unpublished manuscript cited in *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.
 - 16 Goldberg, D.E., & Samtani, M.P. "Engineering Optimization via Genetic Algorithm." *Proceedings of the Ninth Conference on Electronic Computation*. 1986.
 - 17 Singleton, Andy. "Genetic Programming with C++". *Byte*, February, 1994.
 - 18 Koza, J.R. *Genetic Programming II: Scaleable Automatic Programming by Means of Automatically Defined Functions*, MIT Press, Cambridge, MA, 1994.
 - 19 De Jong, K.A. "On Using Genetic Algorithms to Search Program Spaces." *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pp. 210-216. 1987.
 - 20 Kinnear, K.E. Generality and Difficulty in Genetic Programming: Evolving a Sort. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, Ed. San Mateo, CA: Morgan-Kaufmann, pp. 287-294. 1993.
 - 21 Koza, J.R. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.
 - 22 Brooks, Rodney A. "A Robust Layered Control System for a Mobile Robot". *IEEE Journal of Robotics and Automation*, Volume RA-2, no. 1, March 1986.
 - 23 Brooks, Rodney A. "A Robot that Walks: Emergent Behaviors from a Carefully Evolved Network." *IEEE Journal of Robotics and Automation*, Volume CH2750, August 1989.
 - 24 Neubauer, Werner. "Locomotion with Articulated Legs in Pipes or Ducts." *Robotics and Autonomous Systems* 11, 1993.
 - 25 Handey, S. "The Automatic Generation of Plans for a Mobile Robot via Genetic Programming with Automatically Defined Functions." *Advances in Genetic Programming*, Kenneth E. Kinnear, ed. MIT Press, Cambridge, MA, 1994.
 - 26 Spencer, Graham. "Automatic Generation of Programs for Crawling and Walking." in *Advances in Genetic Programming*, Kenneth E. Kinnear, ed. MIT Press, Cambridge, MA, 1994
 - 27 Reynolds, C. W. "Evolution of Obstacle Avoidance Behavior: Using Noise to Promote Robust Solutions." In *Advances in Genetic Programming*, Kenneth E. Kinnear, ed. MIT Press, Cambridge, MA, 1994
 - 28 Rutman, Nathaniel. "Automated Design of Modular Robots." S.M. Thesis, Mechanical Engineering Department, Massachusetts Institute of Technology. 1995.
 - 29 Madhani, A. and Dubowsky, S., "Design and Motion Planning of Multi-Limb Robotic Systems: The Force Workspace Approach," Proc. 1992 ASME Mechanisms Conference, Scottsdale, AZ, Sept. 1992.
 - 30 Madhani, A.J., "Design and Motion Planning of Robotic Systems Subject to Force and Friction Constraints," S.M. Thesis, M.E. Dept, MIT, 1991.
 - 31 Madhani, A.J., Dubowsky, S. "Motion Planning of Mobile Multi-Limbed Robotic Systems Subject to Force and Friction Constraints," *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, 1992.
 - 32 Dubowsky, S., Moore, C., Sunada, C. "On the Design and Task Planning of Power Efficient Field Robotic Systems", *Proceedings of the ANS 6th Topical Meeting on Robotics and Remote Systems*. Monterey, CA, Feb 5-10, 1995.

33 Dubowsky, S, Cole, J., Rutman, N., and Sunada, C., "Mobile Autonomous Robotics Systems for Unstructured Environments-With Applications to the USS Constitution" *Proceedings of the NSF Design and Manufacturing Grantees Conference*, San Diego, CA, January 4-6, 1995. Invited.,

34 Cole, J., Dubowsky, S., Rutman, N., and Sunada, C., "The Application of Advanced Robotics and Sensors to the Preservation of the USS Constitution". *Proceedings of the Conference on the Technical Aspects of Maintaining, Repairing and Preserving Historically Significant Ships*, September 12-14, 1994, Boston, MA.

35 Roach, John Charles. *Old Ironsides: an Essay in Sketches*. Department of US Navy.

7103-78