Abstract of "Interval Programming: A Multi-Objective Optimization Model for Autonomous Vehicle Control" by Michael R. Benjamin, Ph.D., Brown University, May 2002.

Controlling the behavior of a robot or autonomous vehicle in a stochastic, complex environment is a formidable challenge in artificial intelligence. In stochastic domains, both the current state of the vehicle and the environment are typically reconsidered before deciding each action. If the domain is simple enough, effective plans can be encoded by predetermining the best vehicle action for all possible contingencies, or in all possible vehicle states. In complex environments, particularly with other vehicles, the explosion of possible contingencies or vehicle states prohibits this. In these cases, behavior-based architectures are often employed, with each behavior focussed on a specialized vehicle objective. Effective overall vehicle behavior relies heavily on the proper combination, or arbitration, of individual behaviors.

In this work, we present a mathematical programming model, interval programming (IvP), for finding an optimal decision given a set of competing objective functions. We concur with others who believe effective behavior-based action selection involves a multi-objective optimization problem where each behavior contributes a single objective function. To date, such methods have depended on objective functions defined over a sufficiently small discrete decision space as to allow explicit evaluation of all decisions. We believe this is unrealistic in practice and that measures typically taken to sidestep this problem are unacceptable. On the other hand, we also believe that traditional analytical multi-objective optimization methods make demands on objective function form that are unrealistic from the vehicle behavior perspective.

The IvP model strives for a rich balance of speed, flexibility, and accuracy through the use of piecewise linearly defined objective functions. The piece boundaries are typically intervals over decision variables, but may also be intervals over consequences of decision variables coupled with time. This allows behaviors with different levels of planning abstraction to be blended in each decision. The work here is presented in three parts. First we define the IvP model and show how behaviors create IvP functions with sufficient speed and accuracy. Then we provide a set of algorithms for finding quick, globally optimal solutions to the multi-objective IvP problem using branch and bound techniques. And finally, using an underwater vehicle simulator and a group of core vehicle behaviors, we demonstrate the IvP model on the particularly difficult problem of transiting with other moving, potentially uncooperative, vehicles creating time dependent path obstructions.

Abstract of "Interval Programming: A Multi-Objective Optimization Model for Autonomous Vehicle Control" by Michael R. Benjamin, Ph.D., Brown University, May 2002.

Controlling the behavior of a robot or autonomous vehicle in a stochastic, complex environment is a formidable challenge in artificial intelligence. In stochastic domains, both the current state of the vehicle and the environment are typically reconsidered before deciding each action. If the domain is simple enough, effective plans can be encoded by predetermining the best vehicle action for all possible contingencies, or in all possible vehicle states. In complex environments, particularly with other vehicles, the explosion of possible contingencies or vehicle states prohibits this. In these cases, behavior-based architectures are often employed, with each behavior focussed on a specialized vehicle objective. Effective overall vehicle behavior relies heavily on the proper combination, or arbitration, of individual behaviors.

In this work, we present a mathematical programming model, interval programming (IvP), for finding an optimal decision given a set of competing objective functions. We concur with others who believe effective behavior-based action selection involves a multi-objective optimization problem where each behavior contributes a single objective function. To date, such methods have depended on objective functions defined over a sufficiently small discrete decision space as to allow explicit evaluation of all decisions. We believe this is unrealistic in practice and that measures typically taken to sidestep this problem are unacceptable. On the other hand, we also believe that traditional analytical multi-objective optimization methods make demands on objective function form that are unrealistic from the vehicle behavior perspective.

The IvP model strives for a rich balance of speed, flexibility, and accuracy through the use of piecewise linearly defined objective functions. The piece boundaries are typically intervals over decision variables, but may also be intervals over consequences of decision variables coupled with time. This allows behaviors with different levels of planning abstraction to be blended in each decision. The work here is presented in three parts. First we define the IvP model and show how behaviors create IvP functions with sufficient speed and accuracy. Then we provide a set of algorithms for finding quick, globally optimal solutions to the multi-objective IvP problem using branch and bound techniques. And finally, using an underwater vehicle simulator and a group of core vehicle behaviors, we demonstrate the IvP model on the particularly difficult problem of transiting with other moving, potentially uncooperative, vehicles creating time dependent path obstructions.

Interval Programming: A Multi-Objective Optimization Model for Autonomous Vehicle Control

by
Michael R. Benjamin
B. S., Rensselaer Polytechnic Institute, 1988
Sc. M., Rensselaer Polytechnic Institute, 1991
Sc. M., Brown University, 1998

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2002

This dissertation by Michael R. Benjamin is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____

_____
Leslie P. Kaelbling, Director

Recommended to the Graduate Council

Date _____

_____
Stan Zionts, Reader
(State University of New York, Buffalo)

Date _____

_____
Pascal van Hentenryck, Reader

Approved by the Graduate Council

Date _____

_____
Peder J. Estrup
Dean of the Graduate School and Research

Vita

Vitals

Mike Benjamin was born August 16th, 1966 in Rochester, New York. He was admitted in the Computer Science department at Rensselaer Polytechnic Institute in 1984, and graduated in 1988 with a B.S. degree. He was admitted in the department of Philosophy at Rensselaer in 1988 and graduated in 1991 with an M.Sc. degree. He began work, in 1990, as a computer scientist for the Navy in Newport, RI, and continued this work upon entering the Ph.D. program in the Computer Science Department of Brown University.

Education

Ph.D. in Computer Science, May 2002.
Brown University, Providence, RI.

M.Sc. in Computer Science, May 1998.
Brown University, Providence, RI.

M.Sc. in Philosophy, May 1991.
Rensselaer Polytechnic Institute, Troy NY.

B.S. in Computer Science, May 1988.
Rensselaer Polytechnic Institute, Troy NY.

To my brother Dan,

Acknowledgements

I owe a measure of gratitude to several people, without whom this work would have been either impossible or unbearable.

First I would like to thank my advisor, Leslie Kaelbling. From the time I began working with her, the rate of exposure to many new and interesting ideas and perspectives was at times dizzying. She was often times an oracle for determining relevance, and taught me by example how to work efficiently and stay focussed. If I have improved myself at all during this period, it is largely due to her influence.

At Brown, I also owe a huge thanks to John Savage for the many hours he spent with me during the summer of 1998, helping me pull together my disconnected ideas, and teaching me many intangible lessons. To Pascal van Hentenryck, I am grateful for the support during my early years at Brown, the sharing of insights into the world of optimization, and for serving as a committee member. And to Phil Klein, I am a richer person from our many discussions at the white board and the sharing of insights into the world of algorithms.

My life at Brown would not have begun at all were it not for Paris Kanellakis, my first Ph.D. advisor. His memory will stay with me for a lifetime, as will my gratitude for all that he was able to share during our brief two years together.

Vaso Chatzi and Luis Ortiz were, beyond doubt, the best officemates anyone could ask for. Vaso always seemed to have time to listen, and help me sort through whatever confusion I was afflicted with at the moment. I have missed Greek Easter ever since she left, and look forward to someday visiting her bar in Greece. To Thuy Quach, Jose Castanos, Chaoyang Lee, Sonia Leach, Manos Renieris, LE Hartmann, Seewan Eng, Costas Busch, Katrin Schroeder, Gundars Kokts, Marn Yee Lee, Dina Goldin, Greg Seidman, Galina Shubina, and more recently, Shanlenn Birney and Sangeeta Parikshak, thank you for the very valued friendships. To Zhang Laoshi, thank you for letting me sit in on your Chinese classes and providing me with daily glimpses of life outside the world of computer science.

From the Navy, I owe a heap of gratitude to the two individuals who primarily supported this work, Dr. Kam Ng at the Office of Naval Research, and Dick Philips who directs NUWC's independent research program. Without the support of Ann Silva and Steve Naysnerski, my balancing act between Navy and academic life would have been impossible. My work with Tom Viana is at the root

of any and all good ideas that might be found in this work. Tom's friendship and mentoring have helped keep me straight. As has my friendship with Sue Kirschenbaum. I am grateful to have enjoyed access to Mike Keegan's knowledge of the world of UUV's, and John Baylog's and Mike Incze's knowledge of all things Navy. Jim Griffin and Aldo Kusmik have exposed me to another wonderful application of multi-objective optimization. My officemate Mike Walsh has provided valuable moral support as a fellow Latex and Linux sympathizer.

From MIT, I have enjoyed the company and conversation of my officemate and fellow Brown-transplant, Leonid Peshkin. Its unfortunate that our paths crossed when both our lives were so busy. I am also grateful for my friendships with Terran Lane, Sarah Finney, Natalia Hernandez Gardiol, Samson Timoner, Luke Zettlemoyer, and especially Cynthia Thompson who made the initial transition to the AI lab quite comfortable.

Stan Zionts, from the University at Buffalo, went well beyond the call of duty as a committee member, and provided invaluable feedback from our conversations, and careful notes scribbled on early drafts of this work. On both the eve of my thesis proposal, and thesis defense, his arrival from Buffalo had a calming influence that helped me maintain perspective.

Finally, this work would not have been possible without my family: My father and mother, Richard and Trauti, for their constant support, and my grandmother, my Omi, for her support and patience from afar. My wife Kathleen has endured many days of long working hours, and weekends when there were clearly better things to do than work. I cannot thank her enough for her patience and love.

Contents

List of Tables xv

List of Figures xvii

1 Introduction 1

2 Background: Multi-Objective Action Selection 5

2.1 The Behavior-Based Control Architecture 6

2.2 Action Selection Methods 8

2.3 Mathematical Programming and Multiple Criteria Decision Making 14

2.4 Discussion 19

3 The Interval Programming Model 21

3.1 Interval Programming Functions 21

3.2 Interval Programming Problems 25

3.3 Strengths of the Interval Programming Model 28

4 Creating Interval Programming Functions and Problems 33

4.1 Making IvP Functions From Non-IvP Functions 33

4.2 Methods for Collecting Empirical Results 37

4.3 Empirical Results 40

5 Solving Interval Programming Problems 45

5.1 Search Through the Combination Space 45

5.2 The Use of Grid Structures in IvP Solution Algorithms 48

5.3 Using the Grid Structure During Search 54

5.4 Brute Force Search as an Alternative Solution Method 56

5.5 Plane Sweep as an Alternative Solution Method . . . 59
5.6 Empirical Results . . . 60

6 IvP and Autonomous Underwater Vehicle Control 65
6.1 The Vehicle Control Scenario . . . 65
6.2 Behavior 1: Safest Path . . . 69
6.3 Behavior 2: Shortest Path . . . 74
6.4 Behavior 3: Quickest Path . . . 79
6.5 Behavior 4: Boldest Path . . . 79
6.6 Behavior 5: Steadiest Path . . . 80
6.7 Discussion . . . 81

7 Results: Coordinating the Five Vehicle Behaviors 83
7.1 The IvP Vehicle Simulator . . . 83
7.2 Solving a Single IvP Action Selection Problem . . . 84
7.3 Solving a Series of IvP Action Selection Problems . . . 86
7.4 Scenarios with Moving, Maneuvering Contacts . . . 87
7.5 Discussion . . . 91

8 Conclusions and Future Work 93
8.1 Conclusions and Contributions . . . 93
8.2 Future Considerations . . . 94

A Creating a Piecewise All-Sources Shortest Path Function 97

B Case Studies of Three Action Selection Methods 101
B.1 Voting methods . . . 101
B.2 Action Maps . . . 102
B.3 Fuzzy methods . . . 103

C The Integer, Nonlinear and Convex Programming Models 107
C.1 Integer Programming . . . 107
C.2 Nonlinear Programming . . . 109
C.3 Convex programming . . . 109

D Executables and File Types 113

D.1 Creating Ring Functions and IvP Functions 113
D.2 Creating and Solving IvP Problems 117
D.3 Collecting and Reporting Results 119

E Notes on Empirical Results for IvP Function Creation **121**
E.1 Conditions Common to All Experiments 121
E.2 Experiment 1: Accuracy vs. Resources vs. Time 121
E.3 Experiment 2: Linear vs. Constant w.r.t. Dimension Size 123
E.4 Experiment 3: Uniform vs. Non-uniform Pieces 126

F Notes on Empirical Results for IvP Problem Solutions **131**
F.1 Conditions Common to All Experiments 131
F.2 Experiment 1: Plane Sweep Search vs. IvP Methods 131
F.3 Experiment 2: Solution Time vs. Number of Dimensions 132
F.4 Experiment 3: Solution Time vs. Number of IvP pieces 133
F.5 Experiment 4: Solution Time vs. Number of Objective Functions 133

⋆ Parts of this thesis were previously published as Benjamin (2000b,a)

List of Tables

2.1 Case one: a two-element decision space and objective space. 18
2.2 Case two: a two-element decision space and objective space. 18

6.1 Given ownship position and contact solution. 67
6.2 The three decision variables, their domains, and resolution. 68
6.3 Relationship between functions with respect to stability and build time. 78

7.1 A break-down of the created and solved IvP problem. 85

D.1 (a) Usage for the executable `makeRNGs`, (b) the .rng file format. 114
D.2 Example ring file with corresponding ring function. 115
D.3 (a) Usage for the executable `solveIPF`, (b) the .ipf file format. 116
D.4 (a) Usage for the executable `makeIPPs`, (b) the .ipp file format. 118
D.5 Usage for the executable: solveIPP 119

E.1 Combined error and average error vs. time (in seconds). 123
E.2 Combined error vs. number of dimensions. 124
E.3 Average error vs. number of dimensions. 125
E.4 Creation time vs. number of dimensions. 125
E.5 Creation time and `repErr` vs. piece count. 126
E.6 Combined error vs. piece count - for 3 dimensions. 127
E.7 Combined error vs. piece count - for 2 dimensions. 128
E.8 Average error vs. piece count - for 3 dimensions. 128
E.9 Creation time vs. piece count - for 3 dimensions. 129

F.1 Expanded results for the plane-sweep vs. branch-and-bound experiment. 132
F.2 Expanded results for the solution time vs. number of dimensions experiment. . . . 132

F.3 Expanded results for the solution time vs. number of IvP pieces experiment. 133
F.4 Expanded results for the solution time vs. number of objective functions experiment. 134

List of Figures

2.1 The SPA and behavior-based control loops. 5
2.2 General subsumption architecture from Brooks (1986). 7
2.3 A particular behavior-based implementation: DAMN (Rosenblatt, 1997). 7
2.4 A taxonomy for action selection methods. 8
2.5 No-compromise methods. 9
2.6 Single fusion methods (e.g. vector addition). 10
2.7 Multifusion methods. 11
2.8 Maintaining receiver angle as vehicle turns. 11
2.9 Optimizing two dependent variables. 12
2.10 Three analytical functions, $f(x)$, $g(x)$, and $f(x) + g(x)$, and their values at 20 points. 13
2.11 Taxonomy of action selection methods with references. 14

3.1 A non-IvP function, defined on the right and rendered on the left. 22
3.2 An IvP (piecewise defined) function, rendered and defined using 1500 pieces. 22
3.3 Uniform, discrete decision variables *do not* imply uniform pieces. 23
3.4 Piecewise linear vs. piecewise constant functions. 24
3.5 A rectilinear piece with a constant vs. linear interior function. 24
3.6 Rectilinear piecewise vs. non-rectilinear piecewise functions. 25
3.7 Non-rectilinear IvP pieces. 25
3.8 Tightly coupled vs. loosely coupled set of functions. 28
3.9 IvP is fast, flexible and accurate by taking the best of three types of methods. . . . 29
3.10 Two poor ways, (b) and (c), to counter-act a large decision space (a). 30
3.11 Using a piecewise representation, (b), to counter-act to a large decision space (a). . . 30

4.1 Components of the basic control loop and relative computation loads. 34
4.2 The general algorithm for building an IvP function. 34

4.3 Capacity for accuracy and error propagation with respect to dimensions. 36
4.4 Example ring functions with one (a), (b), two (c) and five (d) rings. 38
4.5 Three types of ring functions based on varying parameter inputs. 39
4.6 The average, worst, and combined error for a single experimement sampling 5000 points. 40
4.7 Accuracy vs. time while varying piece count and number of sample points. 41
4.8 Accuracy vs. number of dimensions for piecewise linear and piecewise constant functions. 42
4.9 Error vs. piece count in 8D for piecewise linear and piecewise constant functions. . 43
4.10 Accuracy vs. piece count for uniform-constant, uniform-linear and non-uniform-linear IvP functions. 44

5.1 Intersecting two 2D rectilinear pieces. 46
5.2 The Search tree for $k = 3$ objective functions with m pieces each. 46
5.3 General structure of Recursive `IPAL`. 47
5.4 A bit smarter (and great deal more effective) version of `RIPAL`. 47
5.5 For a given node: Which children intersect? Upper bound on best leaf? 48
5.6 Example grid with 9 grid elements and 23 rectilinear pieces. 49
5.7 Three different universes result in three different grid layouts with the same requested grid element size. 50
5.8 Retrieving intersection information from a grid given a query box. 51
5.9 Finding an upper bound using the grid structure. 52
5.10 Aligning the grid with the initial uniformity of an IvP function. 54
5.11 Versions of `IPAL` and `RIPAL` utilizing intersection information from a grid. 55
5.12 New version of `RIPAL` utilizing the bounding information from a grid. 56
5.13 The algorithm for iterating through the decision space. 57
5.14 Evaluating a point in the decision space w.r.t. the k objective functions. 58
5.15 The algorithm for brute force search through the decision space. 58
5.16 A set of rectangles and its corresponding intersection graph. 59
5.17 Imai-Asano plane sweep method. 59
5.18 The plane sweep algorithm vs. branch and bound. 61
5.19 IvP solution time vs. number of dimensions. 61
5.20 IvP solution time vs. number of pieces in each IvP function. 62
5.21 IvP solution time vs. number of objective functions. 63

6.1 An IvP problem is created and solved on each iteration of the control loop. 66
6.2 The Scenario with ownship and moving contact. 67
6.3 Position after $\langle x_c, x_s, x_t \rangle$. 69

6.4 A collision-avoidance metric based on closest-point-of-approach distance. 71
6.5 (a) A particular situation (b) The resulting $f_{\text{IvP}}(x_c, x_s, x_t)$ objective function, and (c) A different $f_{\text{IvP}}(x_c, x$
6.6 Creation time and accuracy vs. piece count for six different uniform piecewise functions. 73
6.7 A bathymetry function for a region near Florida and Grand Bahama Island. 74
6.8 Determining reachability, for a given depth, using bathymetry data. 75
6.9 Non-uniform representations of $\texttt{bathy}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$. 75
6.10 $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ for a particular region, depth, and destination. 76
6.11 $f_{\text{IvP}}(x_c, x_s, x_t)$ for a particular ownship position and underlying $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$. . . . 77
6.12 Two key linear functions in determining detour distance. 77
6.13 Shortest path and alternative near-shortest paths. 78
6.14 The relationship between speed and utility for the quickest-path behavior. 79
6.15 The relationship between time and utility for the boldest-path behavior. 80
6.16 The relationship between course-change and utility for the steady-path behavior. . . 80

7.1 Screen snapshot of the IvP vehicle simulator. 84
7.2 Resulting solution vector to a single IvP problem instance. 85
7.3 Ownship slows down and cuts behind the moving contact. 86
7.4 IvP solution time for each control loop iteration. 87
7.5 Ownship uses its speed advantage to cross the bow of contact. 88
7.6 Ownship reacts to the contact's course change by slowing and cutting behind the contact. 89
7.7 Ownship reacts to the contact's course change and adjusts its path to the destination. 90
7.8 Ownship reacts to the contact's speed change by slowing and cutting behind the contact. 91

A.1 Identifying direct-path pieces in the initial stage of building $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$. 97
A.2 Identifying the frontier in intermediate stages of building $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$. 98
A.3 The all-sources shortest path algorithm. 99
A.4 Refining the shortest-path distance with help from a neighbor. 99

B.1 Turn radius and camera field-of-regard choices (Rosenblatt, 1997, p.65, 71). 102
B.2 An action map to maneuver a robot *to* an object (Riekki, 1999, p. 52). 102
B.3 Combining actions. (Riekki, 1999, p.43) 103
B.4 Avoid object, and hallway following behaviors (Saffiotti et al., 1999, p. 188) 104
B.5 Behavior combination. (Saffiotti et al., 1999, p.195) 104
B.6 An alternative "avoid object" behavior. 105
B.7 Defuzzification methods. Yen and Pfluger (1995, p.14) and Pirjanian and Mataric (1999, p.6) 106

C.1 The integer vs. fractional optima. 108
C.2 Convex vs. Unimodal functions. 110
C.3 Nonconvexity arises when adding two unimodal functions. 111

D.1 Making ring functions and then IvP functions based on them. 113
D.2 Making IvP problems, solving them, and collecting the results. 117

Chapter 1

Introduction

A common motivation for developing robots, or autonomous vehicles, is the desire to send them into environments where it is less desirable to send humans. For various reasons, these same environments can also make it difficult for a human to control the vehicles remotely. Full autonomy requires the vehicle to contain sufficient models about itself and its environment and to make sufficiently advantageous control decisions to improve its chances of successfully completing its task.

The work presented here was originally motivated by a similar problem of modeling a human decision maker controlling an underwater vehicle, for the purpose of providing a on-line decision aid to the human (Benjamin et al., 1993). In this case, the human was a captain of a U.S. Navy submarine, considered to be an expert in his domain. Yet it was thought that, in certain complex situations, a properly constructed decision model could yield insights providing a tactical advantage. To date, no such general navigation model exists, primarily due to the complexity of the multiple considerations simultaneously juggled by the captain, and the complexity and uncertainty in his environment. The same modeling challenges exist in controlling unmanned vehicles, but with different stakes involved since there are no humans aboard. In certain cases, especially with teams of vehicles, some vehicles may even be considered expendable.

Roughly speaking, the challenges facing autonomous vehicles in real-world domains can be thought of as being composed of three parts: knowing enough about what is going on in the environment, making sufficiently effective decisions based on this knowledge, and making these decisions sufficiently fast, in line with the pace of change in the environment. In this work, we are primarily concerned with the latter two parts. However, a primary goal of this research is to give a vehicle the ability to make progress in multiple competing objectives simultaneously. Since the quality of information obtained by a vehicle often depends on its actions, we are addressing the first part as well.

One common way to ensure rapid, effective vehicle actions is to pre-compute the most effective action for each possible state the vehicle might be in. Effective plans, or sequences of actions, are embedded in the collective pairings of states with actions. This approach, however, is less viable

when the state space is too large to manage. Such is the situation in the case of a vehicle operating in the presence of other vehicles, each with its own unique capabilities, trajectory, and relative position to the vehicle being controlled.

The behavior-based control architecture has been found by many to be a viable architecture in situations where the environment is too complex for a single comprehensive world model, and the ability to react quickly to changes in the environment is especially important. (Brooks, 1986; Rosenblatt, 1997; Arkin, 1998; Pirjanian, 1998; Riekki, 1999). Each behavior specializes in one particular aspect of the overall task of the vehicle, thereby simplifying the implementation of each behavior. Unfortunately, the behaviors often disagree on what next action is best for the vehicle and a workable arbitration scheme therefore becomes necessary. A position taken in our work is that simple arbitration schemes that either suppress all but the most important behavior, or merely average or convolve a single action from multiple behaviors, are schemes that lead to unacceptable shortcomings in overall vehicle behavior. This position is also shared by Rosenblatt (1997, p. 36, 44), Pirjanian (1998, pp. 44-45), and Riekki (1999, pp. 26-27).

Effective coordination of behaviors requires each behavior to not only produce the one action that best serves the goals of the behavior, but also produce alternative actions that may lead to the best overall compromise between behaviors. By rating all possible actions, the behavior is in effect producing an objective function over the domain of all possible vehicle actions, and action selection then can be viewed as a multi-objective optimization problem. Unfortunately, the action domain grows exponentially with respect to the number of action variables, and requiring each behavior to explicitly rate all actions becomes computationally infeasible in all but the simplest applications.

Ideally, we would like each behavior to instead produce its objective function in a concise expression without distorting the true preferences of the behavior. And, ideally, it would be great if there were a method for finding the best compromise decision - regardless of the type of expression produced by the behavior. After all, we would like our method for combining objective functions to be independent of which behaviors are participating at the moment, or may be added in the future. The aim is for the behaviors to work in a plug-and-play, or open-systems manner. Unfortunately, the behaviors that guide such fundamental vehicle tasks such as path following and collision avoidance often produce objective functions that are nonlinear and non-convex, and in general, not amenable to conventional optimization techniques.

The work presented in this thesis aims to strike a balance between these two approaches to combining objective functions: on the one hand there is the explicit evaluation of each action resulting in an overall method that is applicable in all cases but is too slow, and on the other hand there is the implicit evaluation of each action using analytical techniques resulting in an overall approach that is quite fast but applicable only in limited cases. We propose an approach where the objective function produced by each behavior is instead composed in a piecewise linear manner. The number of pieces is far less than the number elements in the action space, and within each piece the linear function implicitly rates all actions contained in the piece. We call this form

of objective function representation, coupled with the algorithms for solving the resulting multi-objective optimization problem, the interval programming model (IvP).

By using piecewise linear functions, there is a price in accuracy with respect to representing the true preferences of the behavior. However, inaccuracies are introduced in making the previously mentioned alternatives work in practice as well. We contend that IvP is an alternative model that offers a unique balance between speed, accuracy and flexibility that is particularly suitable to the domain of autonomous vehicle control. Using a piecewise linear function to approximate an underlying function is certainly not new, and there is a significant body of existing work dedicated to doing this well. But composing a multi-objective optimization problem as a collection of such functions, and the corresponding solution algorithms, are unique contributions of this work.

The IvP solution algorithm, in a nutshell, searches through combinations of pieces from each objective function, once it is produced from its corresponding behavior. A key idea is that the pieces in each function are not expected to be uniform, or consistent in their non-uniformity, between behaviors. Each behavior likely has unique concerns about vehicle control and therefore will produce piecewise functions where both the piece shape and distribution of pieces could be quite different from functions produced by other behaviors. Allowance for this difference is important for behavior modularity as well as effectiveness in accurately representing behavior action preferences.

An important side-effect of the IvP model is the ability to introduce "time" as a decision variable along side the variables corresponding to vehicle actuators. In general, the addition of any additional decision variables represents exponential growth in the size of the decision/action space. In other multi-objective optimization approaches to action selection, such as Rosenblatt (1997) and Riekki (1999), where explicit evaluation of each decision is performed, this growth in the search space is an unwelcome and perhaps unbearable burden. In Rosenblatt (1997), for example, the decision space was "flattened", i.e., uncoupled decisions made for each variable, due to the so-called "curse of dimensionality". In the IvP model, additional decision variables indeed introduce more complexity, but overall solution complexity is tied more to the number of pieces from each function, rather than the dimensionality of each piece.

By allowing "time" to be used as a decision variable, behaviors that rate *consequences* of actions can be merged with ones that rate actions directly. In vehicle navigation, there are situations where the same consequence can be the result of perhaps many different combinations of actions. A distance traveled, for example, can be the result of different vehicle speeds each with different durations of time at that speed. We contend that the ability to reason about the duration of actions is essential in controlling a vehicle amidst other moving vehicles or objects.

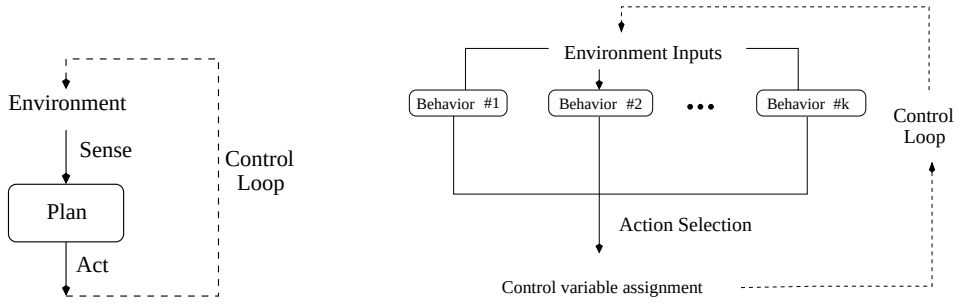
The rest of this thesis is organized as follows: In Chapter 2 background on behavior-based control, action selection methods, and multiple criteria decision making is provided. In Chapter 3, the IvP model is introduced and defined. In Chapter 4, the process of creating IvP functions is discussed. In Chapter 5, the algorithms for solving IvP problems are provided. In Chapter 6, the underwater autonomous vehicle problem is discussed and five vehicle behaviors are provided.

In Chapter 7, results are provided from the IvP vehicle simulator using the five vehicle behaviors provided in the previous chapter. Finally, in Chapter 8, some open problems are provided, and overall results and conclusions are discussed.

Chapter 2

Background: Multi-Objective Action Selection

The aim of this chapter is to introduce the background of the interval programming model (IvP) which touches on the topics of behavior-based control, the action selection problem, and multi-objective optimization. Collectively, we refer to these topics as multi-objective action selection. The behavior-based control architecture can be thought of as an alternative to the traditional sense-plan-act (SPA) architecture shown in Figure 2.1(a). The general idea of the SPA architecture is that control is composed of the three indicated tasks, with a single global model of the world. The model is updated during the sensing phase, and is used by the planning engine to generate the next robot or vehicle action, which then causes a change in the environment. The notion of environment includes not only the other physical objects in the world, but the position and internal states of the vehicle as well.



(a) The sense-plan-act control loop

(b) The behavior-based control loop

Figure 2.1: The SPA and behavior-based control loops.

The primary departure from this in the behavior-based architecture is that individual behaviors

split up the task of sensing the environment and deciding what to do next. The action selection problem is the problem of deciding what next action is best overall for the vehicle, given the inputs of each behavior, which typically disagree in their preferred actions. The role of multi-objective optimization comes from the belief that the action selection problem is best handled when behaviors produce alternative actions to the action selection process, in addition to their single preferred action. When all actions are rated by each behavior, on each iteration of the control loop, the behaviors are effectively producing objective functions. The interval programming model is a particular format for representing these functions, as well as a method for quickly carrying out the action selection process once all functions are present. The remaining sections of this chapter provide more detail on the three topics of behavior-based control, action selection, and multi-objective optimization.

2.1 The Behavior-Based Control Architecture

An agent designed to operate in the real physical world, must deal with the aspect of unpredictability in the environment. An unpredictable event may be something that is out of an agent's control (a car turning into it's path) or an event that is simply left out of the agent's reasoning scope (an on-schedule train crossing it's path). The ability to react to unforeseen events makes life simpler for the agent by requiring it to predict less about the world and allowing it to avoid forming contingency plans for perhaps countless possible deviations from a long term plan. The interest in behavior-based robotics is tied directly to the interest in building robots or agents that are to be situated in such dynamic environments.

In the opinion of Brooks (1991a), traditional AI at that time was poorly prepared to make the jump to building embodied agents situated in dynamic, uncontrived environments. His view was that the field was dominated by a preoccupation with search techniques which in turn promoted an "abandonment of any notion of situatedness" since simulated environments were sufficient for test and evaluation (Brooks, 1999, p. 145). These problem-solving search systems also relied on the use of a complete world model, from which the search algorithms would derive their control actions. Brooks (1999, p. 152) believed such "complete objective models of reality are unrealistic" and that use of these methods in situated agents was therefore also unrealistic.

The subsumption architecture, introduced in Brooks (1986), is shown in Figure 2.2 in its general form. To overcome slow reaction times, at each layer (or behavior), there is a tight coupling between sensors and actuators, and a commitment is made only to the next action. A lower, fundamental behavior such as avoiding obstacles, could take quick control, suppressing or ignoring higher level behaviors. To overcome the pitfalls of relying on a complete world model, each behavior is responsible for taking whatever it needs from the environment for its own limited world perspective. No communication or model sharing is proposed between behaviors. The decentralized nature of the model also contributes to a modular design process and robust performance, and has the effect that overall intelligence attributed to the agent tends to be seen as an emergent property of performance,

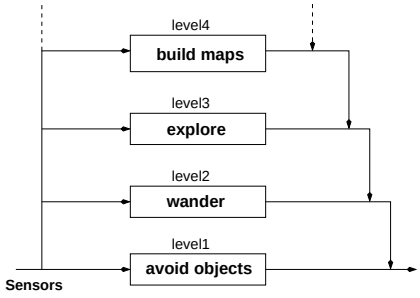


Figure 2.2: General subsumption architecture from Brooks (1986).

rather than due to a single grand design.

The layered design in Figure 2.2 implies a prioritization of behaviors and thus a policy for choosing an action when different behaviors are in conflict. Control of the vehicle alternates between behaviors while the action preferences of other behaviors are ignored completely. This can be viewed as a serious shortcoming if the desire is for the agent to react simultaneously to more than one event, or react in a way that is consistent with longer term plans. In Arkin (1989b); Payton et al. (1990); Maes (1990), this issue came into focus, and cooperative action selection, where an action can be influenced by two or more behaviors simultaneously, have since been common. Behavior-based implementations are typically thought of as containing a separate, distinct component responsible for action selection. As an example, the distributed architecture for mobile navigation (DAMN) proposed in Rosenblatt (1997), and shown below in Figure 2.3, contains an arbiter that receives *votes* from each behavior.

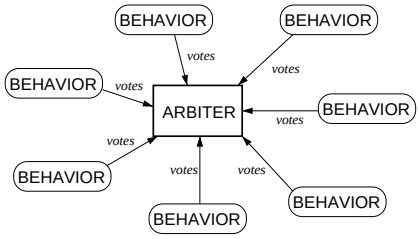


Figure 2.3: A particular behavior-based implementation: DAMN (Rosenblatt, 1997).

Although action selection typically is the only source of communication (albeit one-way) between behaviors, there must be some degree of uniformity in how they communicate to the action selection process, since behaviors are likely to be heterogeneous in their implementation. In the DAMN architecture, this communication comes in the form of the weighted votes over a common set of action choices known to all behaviors.

In general, there is a tradeoff to be made between the complexity of the information being passed from the behaviors and the complexity burden put on the action selection method. On one hand, by emphasizing *quick* decisions, there is the risk of sacrificing the quality of the decision. On the other hand, by emphasizing *quality* decisions, there is the risk of sacrificing the quickness of the decision. In the following section (Section 2.2), the influence of this complexity tradeoff on different classes of

action selection methods will be discussed.

A strength of the behavior-based architecture is its modularity and ability to accept additional behaviors into an existing system without redesigning existing behaviors. Opinions have varied, however, as to what constitutes a behavior. In Brooks (1986), behaviors were devoid of any internal model or representation of the world, and behaviors were thought of as direct mappings from sensory input to actions. Brooks (1989, pp. 449-450) advocated using the world itself as the only model utilized by behaviors. This has contributed to a common perception that anything with a behavior-based flavor is devoid of any internal state representation.

The work presented here does not make a stand on this issue, but addresses rather the output format of behaviors, and the process of action selection. In fact, the behaviors provided in Chapter 6 for our simulated AUV not only contain internal state, but also contain a significant planning component. In Gat (1998), planning processes are delegated to a layer above behaviors with the reasoning being that planning processes typically have a duration that spans across many iterations of the control loop. We contend that such an architecture commitment is unnecessary, and demonstrate so in Chapters 6 and 7. The issues regarding behavior implementation that have been important to us in this work are simply the following. First, behavior implementation must be independent of other behavior implementations. Second, behaviors should not be coerced by the action selection method, for the sake of making action selection easier, into producing anything but its true action preferences. Likewise, the action selection method should not need to be altered as the participation of behaviors change as time or circumstances evolve.

2.2 Action Selection Methods

In this section, our version of a taxonomy for action selection methods (ASMs) is provided, as shown in Figure 2.4. For other taxonomies from different perspectives, see Pirjanian (1998, p.21), Saffiotti (1997), or Pirjanian and Mataric (1999). We mentioned earlier that, in choosing a suitable action

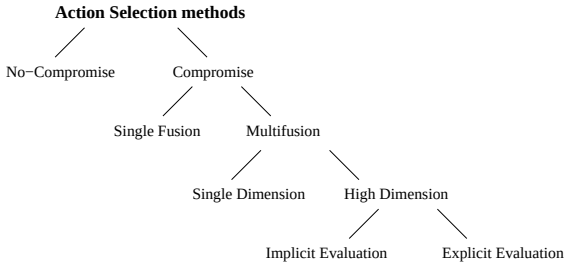


Figure 2.4: A taxonomy for action selection methods.

selection method, there are considerations for both quickness and correctness. By quickness, we simply mean the ability for the ASM to process the behavior inputs quickly. There are two parts to the correctness issue. First, the ASM must faithfully produce the optimal decision given the outputs of the behaviors. Second, the ASM must not coerce a behavior into producing an output that does

not truly reflect the action preferences of the behavior. For example, if an ASM required a linear objective function from each behavior, correctly combining linear functions can be done quickly, but a behavior's actual action preferences may need to be distorted to meet the linear objective function requirement. In our taxonomy, each branch in the tree can be viewed as a lean toward quickness to the left, and a lean toward correctness on the right. The IvP model lies in between the rightmost two leaves. Our motivation is the desire to achieve the best balance of quickness and correctness.

2.2.1 No-Compromise Methods

The simplest way to deal with conflicting behaviors is to let a chosen single behavior determine the next action. The question then is to determine *which* behavior is next to have control. These methods can be considered non-compromising since the concerns of all but the one chosen behavior are completely disregarded. As depicted in Figure 2.5, if two behaviors each declare their pre-

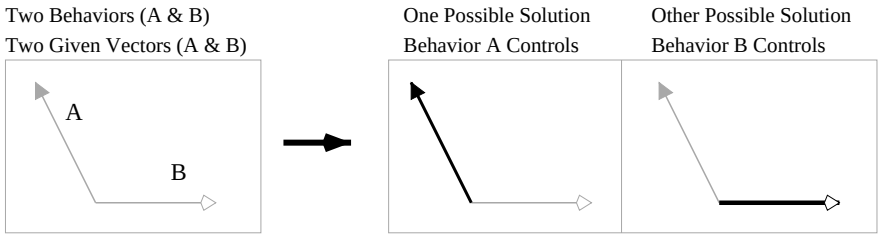


Figure 2.5: No-compromise methods.

ferred action (decision vectors A and B) to the action selection method, then the next action will be one of the two vectors. Brooks' subsumption architecture (Brooks, 1986) used this approach. Others include Maes (1989), Kosecka and Bajcsy (1994), Gat (1998), Roeckel et al. (1999), and Bennet and Leonard (2000). In Brooks (1986), the pre-determined prioritization of behaviors always presents a clear winner among active behaviors. In Gat (1998, p. 201), a higher level "sequencer" determines which behavior should be used at a given time.

The advantage of this approach is that, beyond deciding the relative importance of each behavior, it is trivial for the action selection method to derive a decision vector. Furthermore, each behavior need only provide *one* decision vector since no compromises will be made. The drawback is that, by not compromising, global optimality (actually *any* global perspective) is sacrificed. By this we mean a decision vector that is satisfactory to multiple behaviors (including the highest priority behavior) may be overlooked simply because it wasn't the first choice for the top behavior. For a further discussion of non-compromising methods, see Pirjanian (1998, pp.22-30).

2.2.2 Compromise by Single Fusion

The disadvantages of non-compromising solutions make it tempting to search for a method that retains the simplicity of non-compromising methods while capitalizing in the situations where there

are clear advantages to compromising. By taking a single decision vector from each behavior, a simple compromise can be reached between two conflicting behaviors by creating a new vector that is some combination of the two. We refer to such methods as *single fusion* methods since they convolve, or fuse, a single action from each behavior. As shown in Figure 2.6, if two behaviors each

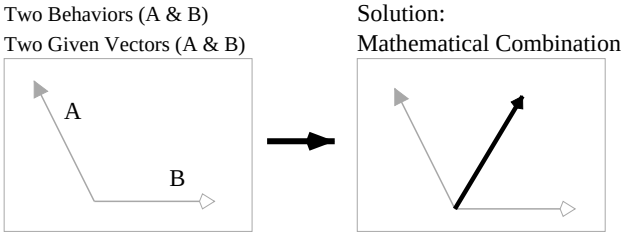


Figure 2.6: Single fusion methods (e.g. vector addition).

declare their preferred action (decision vectors A and B) to the action selection method, then the next method may be a mathematical combination of the two vectors.

The advantage of this approach is that a bit of compromising *is* achieved at a relatively low cost of introduced complexity. The method has had nice results in some applications, most notably through the use of "potential fields" (Khatib, 1985) and "motor schemas" (Arkin, 1987, 1989a, 1992). The disadvantage is that taking a combination of the two vectors may not be appropriate in many circumstances, resulting in a decision that performs poorly for both behaviors. The problem stems from the very limited amount of information offered by each behavior. With one vector, it is impossible to determine *how* to compromise effectively. It is unlikely that any one way of mathematically combining vectors will be appropriate for all circumstances.

2.2.3 Single Dimension Multifusion

To enable behaviors to compromise effectively and reliably, they must also provide alternative decision vectors. We refer to any such method as *multifusion* action selection methods. Compromising between behaviors is done by considering mutually agreeable alternatives and choosing one that comes with the highest combined backing from the contributing behaviors. This means the behaviors also typically indicate the degree to which alternatives are preferred to others. As shown in Figure 2.7, if two behaviors each declare their preferred action (decision vectors A and B) to the action selection method, along with a list of rated alternatives, then the next action will be the best prioritized combination of alternatives. The advantage of multifusion is that the action selection has a more global perspective from the relative wealth of information provided by the behaviors.

In a typical robot or autonomous vehicle application, an action may be comprised of more than one control variable, such as course and speed, or rudder-angle and acceleration. In some cases, the next action can be determined by setting the value for each variable separately, in sequence. When action selection is done in this manner, with behaviors producing action preferences separately for some subset of the control variables, we call this *single dimension multifusion*. If the variables

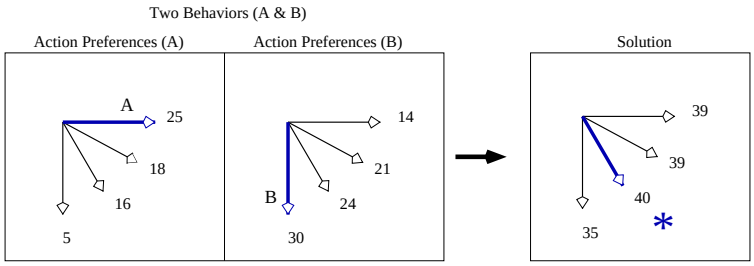


Figure 2.7: Multifusion methods.

are completely unrelated, then performing action selection in a sequence of unrelated decisions can be done without harm, perhaps done in parallel or by separate action selection methods. If the variables are indeed related, or dependent on each other, then action selection may proceed by using the information from one variable setting to costrain the following variable settings. In some cases, this is reasonable. Consider the situation shown in Figure 2.8, where the objective is to keep the

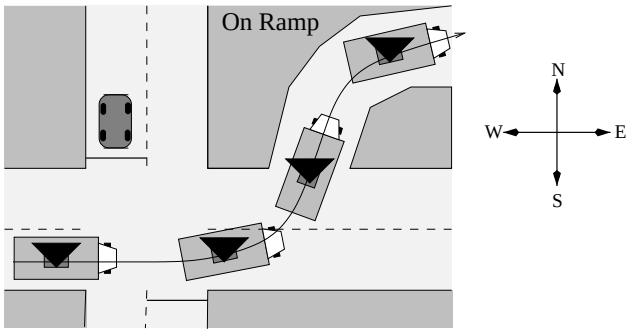


Figure 2.8: Maintaining receiver angle as vehicle turns.

satellite on the roof of the vehicle pointed north. The angle of rotation on its base is dependent on the orientation of the vehicle, once the variables determining the vehicle orientation have been set, the satellite orientation can be chosen. In Rosenblatt (1997), action selection proceeded in this way, by first choosing a vehicle turn radius, and then picking a speed based on the chosen turn radius and vehicle dynamics.

2.2.4 High Dimension Multifusion

In cases where a vehicle is controlled by more than on action variable, it is quite common for the decision space to be composed of the Cartesian product of each variable's domain, due to vehicle dynamics and objectives. Consider the situation in Figure 2.9, where two behaviors have ranked the possible speed and turn angle settings in the manner indicated on the left, such that higher settings are more optimal in both cases. In the middle, the composition of the two functions is shown in two dimensional space. If these two objectives were the only considerations, one could again select an action by first optimizing one variable and then the other. On the right in Figure 2.9, certain

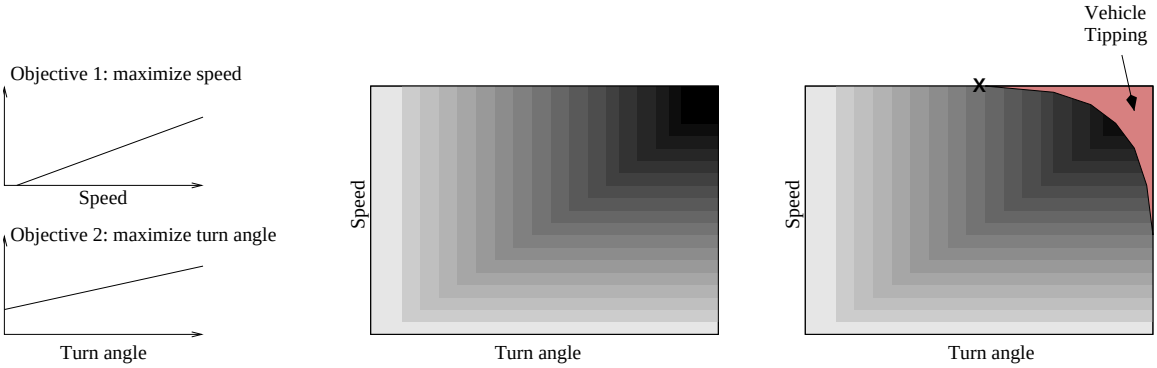


Figure 2.9: Optimizing two dependent variables.

decisions are deemed illegal to prevent the vehicle from tipping at high speeds and large turn angles. If the speed variable were optimized first, the point marked by an **X** in the figure would be the result of making a subsequent constrained choice for turn angle, which is far from optimal. We call action selection methods that search through a decision space composed of the Cartesian product of two or more variables *high dimension multifusion* methods.

Working with a combined variable space introduces a computational burden on the search process. Overcoming this difficulty is a primary motivation of this work, based on the belief that single variable action selection, even when done in a constrained sequence, is insufficient for proper vehicle control in the domains of interest. In Rosenblatt (1997), a good example of single dimension multifusion, he indicated (p. 142) than an important part of his future work was the "coordination of multiple degrees of freedom", but noted that treating the decision space as the Cartesian product of each decision variable was prohibitively expensive. This was for only two decision variables with relatively small domains. In Riekki (1999) the action selection problem is indeed treated as a search through the decision space created by the Cartesian product of each decision variable's domain, but the resulting action is not guaranteed to be optimal. In the next section we address the issue of searching through a large, high dimension decision space.

2.2.5 Explicit vs. Implicit Evaluation of Actions

In searching through a high dimension action space, a method is needed that will not only produce the optimal result, but also is independent of the form of the contributing objective functions. The voting method proposed by Rosenblatt (1997) satisfies these two requirements. To illustrate, consider the situation in Figure 2.10 where there is a small one-dimensional decision space with two behaviors rating individual variable settings as indicated in the left two plots. Each behavior contributes twenty speed-value pairs based on whatever criteria it uses. In the action selection method, if the ratings are tallied by adding, the result is shown on the right. The optimal setting is easy to find after twenty exercises of addition and a check for the largest value. This method is guaranteed to find the optimal action or actions, and is not affected whatsoever by whatever

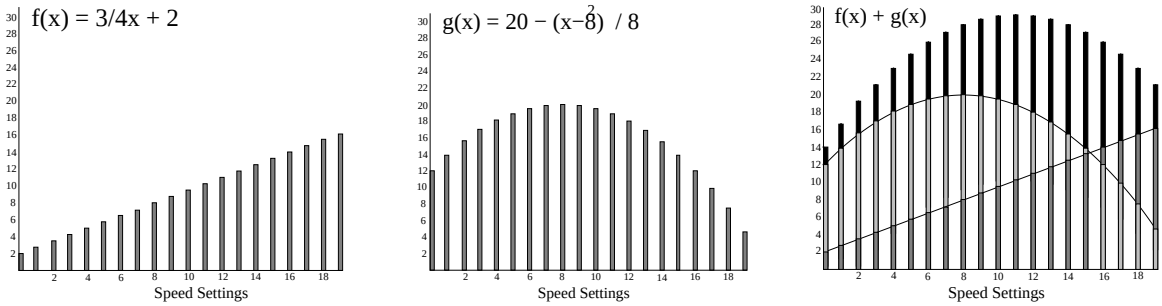


Figure 2.10: Three analytical functions, $f(x)$, $g(x)$, and $f(x) + g(x)$, and their values at 20 points.

underlying function the ratings are derived from. We say this method is using *explicit evaluation* since each point in the action space is explicitly evaluated.

Now suppose that, instead of twenty possible values for speed, there were a thousand or a million choices instead, perhaps because the resolution on the speed control is incredibly accurate and such precision is relevant to the objectives. The time needed to add the ratings for each possible setting may no longer be acceptable. It is tempting instead to appeal to the underlying form of the two contributing functions. These functions, while not communicated by the contributing behaviors, may have captured the concept used by the behaviors, to generate the twenty (or million) action-value pairs. Adding these two functions (also shown in Figure 2.10) together results in $f(x) + g(x) = -x^2/8 + 11/4x + 14$. The derivitive of this function $(-x/4 + 11/4)$ has a root at $x = 11$ indicating a global maximum, since both functions are unimodal. This may be much faster than explicitly evaluating a very large number of speed settings. We say such a method is using *implicit evaluation* since a solution is found without explicitly considering all individual actions.

In autonomous vehicle control, we don't expect to encounter control variables with such incredibly large domains, or variables where a discrete domain is inappropriate. However, we do indeed expect our applications to have perhaps several coupled control variables with moderately large domains, resulting in a high dimension action space with an extremely large number of elements. This means that explicit evaluation methods are unlikely to be viable except in limited, simple circumstances. The use of implicit evaluation in a manner similar to the example in Figure 2.10 is extremely dependent on the actual form of the analytical functions. Although it may be an accurarate and quicker method for searching through a high dimension space, this dependency on function form may disqualify it from being used, or may result in unintended errors by coercing a behavior to produce an objective function in a form that is not true to its actual action preferences.

2.2.6 Discussion

Our taxonomy of action selection methods represents a progression from simple and fast to more complex and computationally challenging methods. The position of the interval programming method in this taxonomy, along with some of the other works mentioned, is indicated below in Figure 2.11.

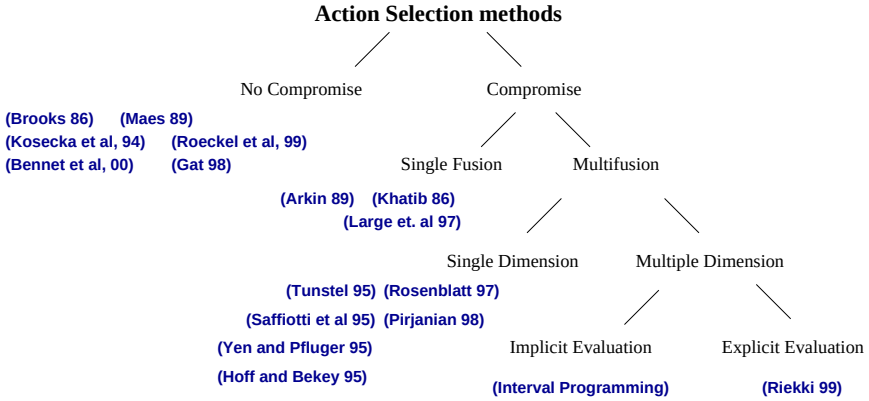


Figure 2.11: Taxonomy of action selection methods with references.

The increase in complexity is perhaps due to the increased expectations of performance put upon the agents in more recent applications. As applications become more ambitious, not only are the expectations higher for flawless performance, but the environments in which they perform become more complex, less predictable, less observable, and less cooperative. The early applications of behavior-based control featured robots performing tasks in a relatively cooperative office environment. In Brooks (1989, p. 437), he notes "Over the course of a number of hours of autonomous operation, our physical robot has not collided with either a moving or fixed obstacle. The moving obstacles have, however, been careful to move slowly."

2.3 Mathematical Programming and Multiple Criteria Decision Making

The general mathematical programming model deals with the optimization of a single objective function and a set of constraints. The objective function is a function mapping each point in the decision space to some value, and the constraints indicate which points in the decision space are considered legal. Although we indicated in the previous section that we are interested in simultaneously optimizing more than one objective function, we first consider the single objective function model for a couple reasons. The first reason is that one way of optimizing a group of objective functions is to combine them into one function, and then apply the techniques applicable to single objective function optimization.

The second reason is that it allows us focus on a key idea that motivated the development of interval programming. The basic purpose of any particular mathematical programming model is to provide a means for accurately expressing a problem, backed by a set of sufficiently fast algorithms for finding a solution. The degree to which the model is useful to people depends on the flexibility of the model to accurately represent a wide set of problems of interest, and the ability of the algorithms to solve sufficiently large problems, sufficiently fast. The details of the solution algorithm should

be largely hidden from the user, and insensitive to the particular problem instance. Certain models will be good for some problems, and inadequate for others. The motivation for the IvP model was to fill a gap where existing models were inadequate for the problems found in multi-objective action selection.

2.3.1 The General Optimization Model

In a mathematical programming problem, the decision-maker's goal is to pick a set of decision variable values that optimizes an objective function, subject to the requirement that the decision variables satisfy certain constraints. This can be expressed as:

$$\begin{array}{lll} \underset{\boldsymbol{x} \in R^n}{\text{optimize}} & f(\boldsymbol{x}) & \text{General Optimization model} \\ \text{subject to} & \boldsymbol{x} \in S, & \end{array}$$

where $f(\boldsymbol{x})$ is the *objective function* over the n decision variables $(x_1, \ldots, x_n)$, and the set S is the set of $feasible$ vectors. The domain of each decision variable here is the set of real numbers, but this may be different for different models. The *decision space* is formed from the Cartesian product of the variable domains. The choices made about the variable domains, the form of the objective function, and the form of the feasible set determine the expressive power of the model, and likewise impact the available solution algorithms.

Linear Programming

Linear programming (LP), introduced by Dantzig (1948), is arguably the canonical mathematical programming model, formed by restricting the objective function to be linear as well as restricting the feasible set to be those vectors satisfying a set of linear inequalities:

$$\begin{array}{lll} \underset{\boldsymbol{x} \in R^n}{\text{minimize}} & c_1 x_1 \; + \ldots + c_n x_n & \text{Linear Programming model} \\ \text{subject to} & a_{11} x_1 \; + \ldots + a_{1n} x_n \leq b_1 & \\ & \ldots & \\ & a_{m1} x_1 + \ldots + a_{mn} x_n \leq b_m & \\ & x_1, \ldots, x_n \geq 0. & \end{array}$$

In using linear programming to model a particular real-world problem, a certain amount of error may be introduced. Consider an example objective function where the variables represent an amount of a particular product to produce, the coefficients represent the profit for each product, and linear constraint functions perhaps represent available labor limits and minimal production requirements.

See the example in Section C.1 in Appendix C. The LP model may only be an approximation because perhaps the market value for the product may go down as production goes up, or over-time labor may be available beyond the stated labor constraint. Sometimes the model may be altered in creative ways to accomodate real-world peculiarities into the linear model, such as, in this case, by adding extra decision variables to represent product produced with over-time labor. In other cases, the linear model remains only an approximation of the real-world situation. However, no model will accurately represent all problems.

The motivation for modeling a problem in line with the linear programming model is the assurance that there exist effective algorithms for finding the optimal solution. These algorithms are not dependent on the variable semantics, and the user need only be sure the real-world problem is represented by a valid instance of the LP model. The linear constraints represent a feasible subregion of the decion space that form a convex polygon. One such vertex will represent at least a tie for the optimal solution. Dantzig's simplex algorithm, in effect, moves from vertex to vertex until a solution is found. The number of vertices may be quite large, and Dantzig himself originally rejected the simplex method due to the unpromising way in which it would search the feasible space (Lenstra et al., 1991). However, in practice, simplex tends to make choices that result in rapid convergence. Specialized variations of the simplex method (such as the *network simplex algorithm* for minimal cost network flow problems) were later developed to capitalize on certain characteristics of particular LP problems.

The linear programming model, and simplex, made its impact primarily due to the fortunate balance between speed and flexibility. The flexibility of the linear format allowed a large and diverse set of problems to be represented, and the simplex method ensured that large problems stated in this manner could be solved quickly. Its worth noting that the arrival of LP occurred in an era when large-scale computing was becoming more available. Prior to the arrival of LP, the idea of an objective function to be extremized was a novel feature, primarily due to its noncomputability. Thus simplex was a practical, viable method, in part, because the search process was not carried out by hand. The World War II and post-war applications also emphasized results, and therefore the theoretical worst-case analysis of simplex came well after the point in time when people found it useful in practice.

Mathematical Programming and Action Selection

In the multi-objective action selection problem, described in Section 2.2, behaviors each produce a single objective function that rate preferred and alternative actions. If these functions are combined, e.g. added, to produce a single function, the result will certainly not be a linear function. Alternative models are discussed in Appendix C that also optimize over a single objective function. We contend, however, that the behavior preferences in vehicle control applications are likely to result in nonlinear, nonconvex combined objective functions. Finding quick, optimal solutions (or sub-optimal solutions with some assurance of proximity to optimal) in such problems are notoriously difficult, thus leaving

a need for a different model to address the multi-objective vehicle control problems of interest to us.

2.3.2 The Multiple Criteria Decision Making (MCDM) Model

The MCDM model differs from the mathematical programming model primarily in that it strives to optimize several objective functions simultaneously as opposed to just one. The general form of the multiple criteria decision making problem is given by Miettinen (1999):

$$\begin{array}{lll} \text{optimize} & f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x}) & \text{General MCDM} \\ \text{subject to} & \boldsymbol{x} \in S & \text{model} \end{array}$$

where there are k objective functions defined over n decision variables. The vector $\boldsymbol{x}$ is a decision or an assignment to the n decision variables. S is the set of the feasible, i.e., legal, decisions. Like the general optimization model with one objective function, the ways in which subclasses are formed from the general model depend on the kinds of objective functions allowed, the feasible region specification, and the domains of the decision variables. In Triantaphyllou (2000), discrete variable domains are used to distinguish MCDM from multi-objective decision making methods characterized by continuous variable domains. We use the terms interchangeably here.

The MCDM model however has an additional factor that separates one model from another, namely the *value function*. The value function rates the collective output of each objective function given a particular point in the decision space as input to each function. The value function is defined over the *objective space*, and captures the preferences and priorities of the decision maker.

The Objective Space and Value Functions

Since the MCDM model contains n decision variables and k objective functions, each n dimensional vector $\boldsymbol{x}$ in the decision space has a corresponding k dimensional vector $\boldsymbol{z} = \langle f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x}) \rangle$ in the *objective space*. Ideally, the best solution to an MCDM problem, $\boldsymbol{z}^*$, would be the rare decision that optimizes all objective functions simultaneously. If each element of $\boldsymbol{z}^*$ were the optimum of the corresponding $f_i(\boldsymbol{x}^*)$, there would be little contention that $\boldsymbol{x}^*$ is the best solution. But in the vast majority of cases, where tradeoffs must be made, an optimal decision cannot be identified unless there is an ordering of the elements of the objective space. To see this, consider a simple MCDM problem with only two feasible solutions, $\boldsymbol{x}'$ and $\boldsymbol{x}''$, and two objectives, f_1 and f_2. Suppose $\boldsymbol{x}'$ optimizes f_1 while performing poorly on f_2. Conversely, $\boldsymbol{x}''$ optimizes f_2, while performing poorly on f_1. Assuming that "10" is optimal for both functions, the situation would look like case one in Table 2.1.

$f_1(\boldsymbol{x}') = 10$ $f_2(\boldsymbol{x}') = 2.5$	$f_1(\boldsymbol{x}'') = 2.5$ $f_2(\boldsymbol{x}'') = 10$
$\boldsymbol{z}' = \langle 10, 2.5 \rangle$	$\boldsymbol{z}'' = \langle 2.5, 10 \rangle$

Table 2.1: Case one: a two-element decision space and objective space.

Which solution is best? There is *no* way to answer this until more information is provided. If, for example, the relative priority of objectives were known, then a reasonable response would be to pick the solution that optimizes the more important objective. Assuming a prioritization making f_1 the top priority, $\boldsymbol{x}'$ would be the "best" solution. But suppose that $\boldsymbol{x}''$, although not performing as well on the highest priority objective, performed almost as well, while still optimizing the other objective, as shown in case two, in Table 2.2.

$f_1(\boldsymbol{x}') = 10$ $f_2(\boldsymbol{x}') = 2.5$	$f_1(\boldsymbol{x}'') = \mathbf{9.8}$ $f_2(\boldsymbol{x}'') = 10$
$\boldsymbol{z}' = \langle 10, 2.5 \rangle$	$\boldsymbol{z}'' = \langle \mathbf{9.8}, 10 \rangle$

Table 2.2: Case two: a two-element decision space and objective space.

In this case the decision maker *might* be inclined to declare $\boldsymbol{x}''$ to be the best solution despite the fact that $\boldsymbol{x}'$ performs better on the higher priority objective.

A *value function*, which removes all such ambiguity as to which decision is preferred to another, is a mapping ($\mathbb{R}^k \rightarrow \mathbb{R}$) from each point in the objective space to a scalar value. A value function is just one way of representing the preferences of the decision maker, and not all MCDM models explicitly use a value function. The *lexicographic method*, alluded to in the above cases, is an example of a solution method operating without an explicit value function. The optimal decision optimizes the highest priority objective while lower priority objectives are successively used to break ties until a single decision remains. No value function is used, but it could be derived from the combination of the priorities and the solution method. Without a value function, or a search method that implies a value function, the only notion of optimality is *Pareto* optimality.

Pareto Optimality and Autonomous Agents

The concept of an optimal solution was introduced by the French-Italian economist Vilfredo Pareto in 1896 (Pareto, 1964). The interesting aspect of a Pareto optimal solution is that its optimality can be determined independent of the value function. A Pareto optimal decision is one where none of the components of $\boldsymbol{z}^*$ can be improved without deterioration to at least one of the other components. To be more precise, from Miettinen (1999, p.11):

> A decision vector $\boldsymbol{x}^* \in S$ is *Pareto optimal* if there does not exist another decision vector $\boldsymbol{x} \in S$ such that $f_i(\boldsymbol{x}) \leq f_i(\boldsymbol{x}^*)$ for all $i = 1, \ldots, k$ and $f_j(\boldsymbol{x}) < f_j(\boldsymbol{x}^*)$ for at least one index j. A vector in the objective space is Pareto optimal if it has a corresponding decision vector that is Pareto optimal.

Pareto optimal solutions are also referred to as *nondominated* solutions, and each has the potential to be the overall true optimal solution (although perhaps not unique), given some particular value function. In *interactive* MCDM methods, if the set of Pareto optimal solutions can be identified, the decision maker can "specify and correct her or his preferences and selections as the solution process continues and (s)he gets to know the problem and its potentialities better." (Miettinen, 1999, p. 131) Narrowing the solution space down to the Pareto optimal set can thus greatly facilitate this process, and can be done without concern about which value function the decision maker will ultimately use.

In autonomous vehicle control, however, there is no human decision maker, and there can be no trial-and-error formulation of the value function. In each iteration of the control loop, an MCDM, or multi-objective optimization problem will be created, solved, output, and forgotten. The behaviors then immediately renew the process to produce new objective functions. In the classification of MCDM methods in Miettinen (1999), this approach is known as an *a priori* method since the value function is determined before the solution process is begun. The interval programming solution method is thus an a priori method, and Pareto optimality will play little role in the IvP solution algorithms. Solutions to IvP problems will indeed be Pareto optimal, but the Pareto optimal set is not identified or generated as an intermediate solution step.

2.4 Discussion

In this chapter we discussed the three areas of behavior-based control, action selection, and multi-objective optimization. In Section 2.1 we reviewed and advocated the use of the behavior-based control architecture, and noted that effective implementation in autonomous vehicle control relied on an effective action selection method. In Section 2.2, we provided a taxonomy of action selection methods and discussed their strengths and weaknesses with respect to flexibility, speed, and accuracy. We ultimately advocated action selection methods that formulate and solve a multi-objective optimization problem. In Section 2.3, we reviewed mathematical programming techniques for single and multi-objective function optimization.

In the next chapter, the interval programming model is introduced as an alternative method for representing objective functions, and solving the resulting action selection problem. Our motivation for IvP is to have a model with the flexibility to represent a wide class of objective functions, and a corresponding set of solution algorithms that produce sufficiently fast answers without sacrificing solution accuracy. We contend that, for behavior-based autonomous vehicle control, the existing alternatives reviewed in this chapter are inadequate in one or more of these regards.

Chapter 3

The Interval Programming Model

The key idea in interval programming is the use of piecewise defined objective functions. The motivation for this is the relative flexibility of these functions in approximating any underlying function. Although there is a cost in precision in representing functions this way, we provide algorithms for guaranteeing global optimality when combining the set of IvP objective functions. In this chapter, IvP functions are introduced Section 3.1, then IvP problems are defined in Section 3.2, and finally the strengths of the IvP model compared to alternative approaches will be discussed in Section 3.3.

3.1 Interval Programming Functions

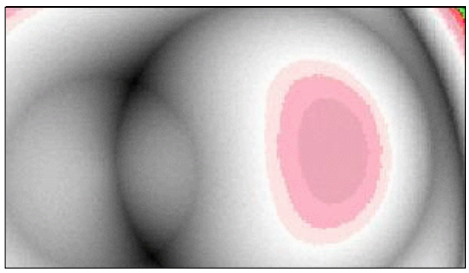
An IvP function is simply a piecewise defined function. Within this characterization there is some room for ambiguity, so before we talk about IvP problems in Section 3.2, we first introduce IvP functions.

Definition 3.1.1 An IvP *function* is piecewise defined such that each point in the decision space is covered by one and only one piece, and each piece is an IvP piece.

Definition 3.1.2 An IvP *piece* is given by a set of intervals, one for each decision variable, and an interior function evaluating each point in the piece.

Each decision variable has a bounded, uniformly discrete domain. Each interval may be defined over a single decision variable, or over a single function, where such a function is defined over one or more decision variables. A piece using exclusively the former such interval is called a *rectilinear* piece. The term *non-rectilinear* piece is used otherwise. In a general (non-IvP) piecewise defined function, the value of the function may be given for certain regions of the domain, with a default value given for all other regions. For example: "$f(x, y) = 25$ if $x \leq 15$ and $y > 0, -25$ otherwise.". This is not allowed in IvP functions, by the two definitions above.

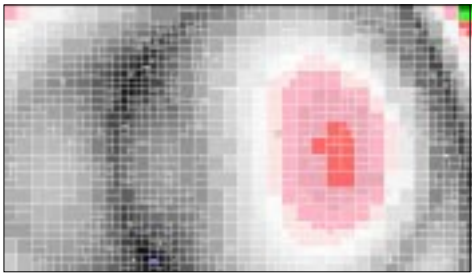
Consider the function shown in Figure 3.1. The function is defined over two variables and is given by the definition on the right. It is rendered on the left, looking down on the x-y plane, where darker points indicate higher values in the function. Note that it is nonlinear and non-convex,



$$
\begin{aligned}
f(x,y) = \quad & ((1 - \tfrac{(\sqrt{(x-150)^2+(y-88)^2}-547)}{1100})^{19} * 200) - 100 + \\
& ((1 - \tfrac{(\sqrt{(x-409)^2+(y-172)^2}-242)}{1100})^{11} * 125) - 100 + \\
& ((1 - \tfrac{(\sqrt{(x-143)^2+(y-145)^2}-149)}{1100})^{5} * 125) - 100
\end{aligned}
$$

Figure 3.1: A non-IvP function, defined on the right and rendered on the left.

but is very concisely stated by the closed-form expression. Compare this with the IvP function in Figure 3.2. It approximates the function in the previous figure by using 1500 pieces with a constant



$$
\begin{aligned}
f(x,y) = \quad & c_1 \text{ if } \quad x_1^- \le x \le x_1^+ \text{ and } y_2^- \le y \le y_2^+ \\
& c_2 \text{ if } \quad x_2^- \le x \le x_2^+ \text{ and } y_2^- \le y \le y_2^+ \\
& c_3 \text{ if } \quad x_3^- \le x \le x_3^+ \text{ and } y_3^- \le y \le y_3^+ \\
& \quad \vdots \\
& c_{1500} \text{ if } x_{1500}^- \le x \le x_{1500}^+ \text{ and } y_{1500}^- \le y \le y_{1500}^+
\end{aligned}
$$

Figure 3.2: An IvP (piecewise defined) function, rendered and defined using 1500 pieces.

function within each piece. It is not as precise, and is considerably less concise in its representation. Nevertheless, the basic idea is that, with enough pieces, intelligently placed with intelligently chosen linear interior functions, the approximation is sufficient.

3.1.1 The Motivations For Using a Uniform Discrete Decision Space

Decision variables in IvP functions are assumed to be bounded and uniformly discrete. This assumption has three motivations. The first is that, in autonomous vehicle control problems, the decision variables typically correspond to actuator settings, such as course or speed of the controlled vehicle. The precision of such actuators have a natural limit such as 0.5°, or 0.1 knots. A decision recommending a new speed of 12.7863 knots is useless, unless the vehicle can confidently execute that request.

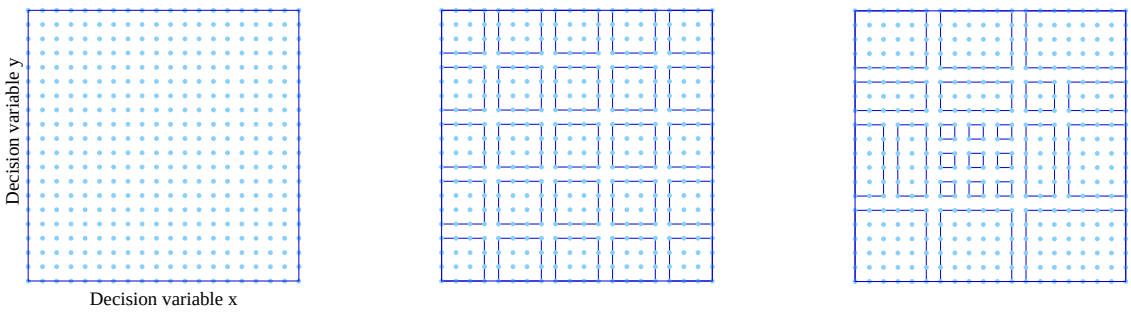
The last two motivations are due to implementation considerations. With a uniform discrete domain, all intervals can by represented using only inclusive bounds. Recall from Definition 3.1.1 that each point in the decision space must belong to at least one IvP piece. If the space were continuous, the set of pieces would need to utilize both inclusive and exclusive bounds to uphold

this property. For example, if the domain $[0, 10]$ is uniformly discrete, containing only integers, it can be represented with two intervals $0 \leq x \leq 5$ and $6 \leq x \leq 10$. If it were continuous, exclusive bounds would be necessary, as in the pair $0 \leq x \leq 5$ and $5 < x \leq 10$. By needing to use only inclusive bounds, each interval, $x^- \leq x \leq x^+$, need only store the upper and lower bound information, and can omit the two other boundary flags that would otherwise be needed to distinguish between $x^- < x \leq x^+$, or $x^- < x < x^+$. Since large numbers of pieces are typically involved in IvP problems, this simplification results in a significant reduction of memory size.

The last consideration is that, with uniform discrete domains, each piece represents a finite number of elements (decisions), making small brute-force calculations possible in limited circumstances. This becomes extremely useful when dealing with pieces having non-rectilinear edges, which will be discussed in Section 3.1.4, and Chapter 5.

3.1.2 A Uniform Decision Space Does Not Entail Uniform Pieces

It is important to note the distinction between uniform discrete variable domains and uniform IvP pieces. In Figure 3.3(a), a 2D decision space is depicted with a uniform discrete domain. In Figure 3.3(b), a piecewise defined function using uniform pieces is depicted. In Figure 3.3(c) a



(a) Uniform discrete domain

(b) Uniform domain with uniform pieces

(c) Uniform domain with NON-uniform pieces

Figure 3.3: Uniform, discrete decision variables *do not* imply uniform pieces.

piecewise defined function using non-uniform pieces is shown. The use of non-uniform pieces allows the use of more pieces in areas where the function is more "interesting", i.e., less amenable to approximation by a linear function. The creation of uniform pieces can be extremely quick, however. The issues of uniformity, precision, and speed, are discussed again with empirical results in Section 4.3.3.

3.1.3 IvP Piece Interior Functions: Constant vs. Linear

There is a basic tradeoff in IvP functions between accuracy and speed. Although we haven't yet discussed IvP problems and their solutions, in a nutshell, more pieces mean longer solution times

but also a greater capacity for accuracy in representing functions. There are ways, however, to both increase the accuracy and the speed at the same time. This is done by using more "powerful" pieces, by either using a better interior function or a better piece boundary.

The simplest way to improve IvP functions is to use linear interior functions within each piece as shown in Figure 3.4. We show in Section 4.3.2 that this is both faster and more accurate. The use

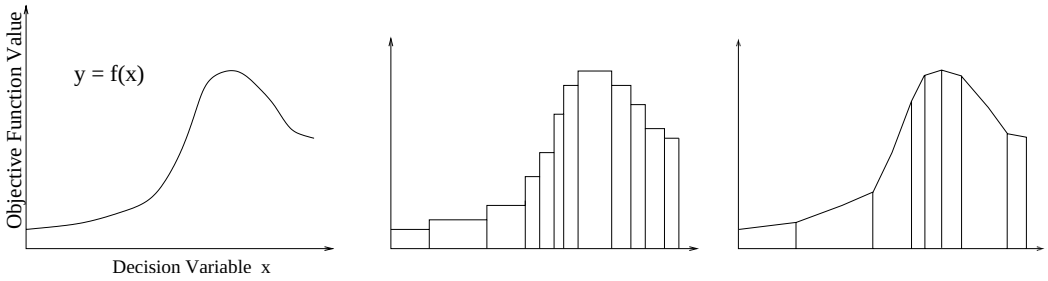


Figure 3.4: Piecewise linear vs. piecewise constant functions.

of piecewise linear functions introduces a bit of new complexity in each step of the solution process, but typically the number of overall steps is drastically reduced, resulting in an overall reduced solution time. There is also more complexity introduced in translating a non-IvP function into an IvP function, as discussed in Chapter 4.

c_i		
x_i^-	$\leq x \leq$	x_i^+
y_i^-	$\leq y \leq$	x_i^+

Piecewise constant

$m_{xi}x + m_{yi}y + c_i$		
x_i^-	$\leq x \leq$	x_i^+
y_i^-	$\leq y \leq$	x_i^+

Piecewise linear

Figure 3.5: A rectilinear piece with a constant vs. linear interior function.

3.1.4 IvP Piece Boundaries: Rectilinear vs. Non-rectilinear

Taking simple intervals over decision variables results in pieces with *rectilinear* edges, i.e., rectangles with edges parallel to the variable axes. In addition to altering the interiors of the pieces, as just discussed in 3.1.3, the use of non-rectilinear edges can also greatly increase the expressive capacity of each piece. As with the use of linear interiors, the aim is to increase the expressive capacity of IvP functions while reducing the solution time in solving IvP problems. This is done by allowing pieces to be composed of intervals not only over the variables, but also over functions of the variables. Consider the example shown in Figure 3.6. On the left, the following function is rendered:

$$f(x,y) = \frac{1}{1 + (\sqrt{(x-h)^2 + (y-k)^2} - m)^2}.$$

A piecewise defined function is used in the middle rendering, using rectilinear pieces. On the right, a piecewise defined function using non-rectilinear pieces is used. In this case, intervals over a function giving the radius is used.

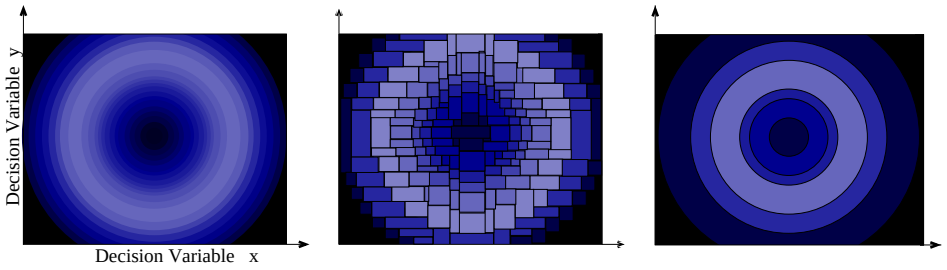


Figure 3.6: Rectilinear piecewise vs. non-rectilinear piecewise functions.

The usefulness of non-rectilinear pieces will be demonstrated in Chapter 6, when objective functions found in vehicle control applications are considered. Objective functions will be used that directly rate different actuator settings like direction and speed, as well as objective functions that rate consequences of actuator settings, such as distance traveled, or resulting positions. The generalization to non-rectilinear pieces requires each piece to contain an interval over each desired function as shown below.

c_i		
x_i^-	$\leq x \leq$	x_i^+
y_i^-	$\leq y \leq$	x_i^+
f_i^-	$\leq f(x,y) \leq$	f_i^+

(a) Non-rectilinear piecewise constant

$m_{xi}x + m_{yi}y + m_{fi}f + c_i$		
x_i^-	$\leq x \leq$	x_i^+
y_i^-	$\leq y \leq$	x_i^+
f_i^-	$\leq f(x,y) \leq$	f_i^+

(b) Non-rectilinear piecewise linear

Figure 3.7: Non-rectilinear IvP pieces.

In the piecewise linear case, a coefficient is kept for each interval, including intervals over functions. Note that the new interval looks like any other interval over a decision variable. The IvP pieces are structured like this so that these non-rectilinear pieces can indeed be treated as rectilinear pieces in the core IvP operations, such as taking the intersection of two pieces.

3.2 Interval Programming Problems

An IvP problem is composed simply of a collection of IvP functions and an associated weight, or priority, for each objective function. Each function is typically a reflection of an objective or goal of the decision maker or autonomous agent, and each priority reflects the relative importance of the goal, given the situation or context.

Definition 3.2.1 An interval programming *problem* consists of a set of k piecewise-defined objective functions. Each objective function, defined over n decision variables $(x_1, \ldots, x_n)$, has an associated priority weight (w_i). The general form is given:

maximize	$w_1 f_1(x_1, \ldots, x_n) + \ldots + w_k f_k(x_1, \ldots, x_n)$
such that	f_i is an IvP piecewise defined function,
	$w_i \in [0, +\infty]$.

The *solution* to an IvP problem, and how global optimality is achieved, is discussed below in Section 3.2.1. Note that the definition given above assumes an additive preference structure. Issues concerning this are discussed in Section 3.2.2. Note also that IvP problems do not contain a feasible region as in the general optimization model and MCDM models given in Sections 2.3.1 and 2.3.2. This issue is discussed more in Section 3.2.3. Finally, "loosely coupled" IvP problems are defined and discussed below in Section 3.2.4 as a significant special class of IvP problems.

3.2.1 IvP Solutions and Global Optimality

A solution to an IvP problem is the single decision $\langle x_1, \ldots, x_n \rangle$ with the highest value, when evaluated by $w_1 f_1(x_1, \ldots, x_n) + \ldots + w_k f_k(x_1, \ldots, x_n)$. Note that the assumption of uniform, discrete variable domains (Section 3.1.1) opens the door to brute force, exhaustive searches of the decision space, and thus "globally optimal" solutions. Of course such searches need to be avoided for practical reasons. If limited to brute force search, time constraints will confine an application to the use of very small decision spaces of one or two variables as discussed in the section on voting methods (B.1) and action maps (B.2).

In IvP problems, there is a second way of viewing the definition of a *solution*. Note that, given the definition of an IvP function, each decision point lies in exactly one piece from each function. So the second way to view a solution is the set of k pieces, one from each function, that each contain the best decision. This means that the "solution space" is the set of all possible combinations of pieces from each function. Admittedly, this space can be much bigger than the decision space, but this second view opens the door to a different kind of search, while still guaranteeing global optimality. This issue is returned to in Section 5.1.

3.2.2 IvP and the Additive Preference Structure

By the definition of an IvP problem, it is apparent that the value function is fully specified and indicates an additive preference structure. As Miettinen (1999, p. 21) points out: "If we had at our disposal the mathematical expression of the decision maker's value function, it would be easy to solve the multi-objective optimization problem. The value function would simply be maximized by some method of single objective optimization". This may cause one to question whether the IvP model is best thought of as an MCDM model or not. This situation is not surprising since the primary motivating application is autonomous vehicle control, where there is no decision maker in the loop, and the agent must simply take its best shot on determining its value function, and then generate an answer as quickly as possible.

In the case where there *is* a decision maker in the loop, the value function, while fully specified

initially, may in fact change as a result of interaction with the decision maker. This is the intended scenario in the cases where IvP is used for simulation-based design applications. As Miettinen (1999, p. 21) goes on to say: "There are several reasons why this seemingly easy way is not generally used in practice. The most important reason is that it is extremely difficult, if not impossible, for a decision maker to specify mathematically the function behind her or his preferences. Secondly, even if the function were known, it could be difficult to optimize because of its possible complicated nature." In this vein, its worth noting that the result of adding two or more piecewise defined functions is not another piecewise defined function, unless the configurations of pieces from each function are identical. Two pieces from two functions, if they overlap, typically do not overlap precisely. As we will see in Section 5.1.2, if we wish to create a single new piecewise defined function from the expression $w_1 f_1(x_1, \ldots, x_n) + \ldots + w_k f_k(x_1, \ldots, x_n)$, this results in a growth in new pieces that is potentially exponential in the dimension of the problem. The bottom line is that, in solving an IvP problem, we do not want to explicitly create a single objective function and then optimize.

There is still some question as to whether an additive preference structure is appropriate. Furthermore, there is nothing about the IvP model or its solution algorithms that precludes a different structure from being supplanted. Care must be made however in cases where two objective functions are derived from two behaviors where the goals have an "either/or" relationship. In this case, the likely remedy is to combine these behaviors into a single behavior.

3.2.3 IvP, Constraints, and a Feasible Regions

A conspicuous feature lacking in the definition of an IvP problem is the *feasible* set $\boldsymbol{S}$, which is a key component in the general optimization and MCDM models discussed in Section 2.3.1 and Section 2.3.2. A feasible set ensures that decisions that go beyond being undesirable, and are outright not permissible, or even nonsensical, are never solution candidates. The feasible set is also, based on its form, a way of guiding the search for the actual solution. In the simplex method for linear programming, for example, the solution process progresses from one vertex of the feasible space to another. But as useful as this may be, assumptions about a feasible region within a model may disqualify its use entirely. For example, discontiguous, or non-convex feasible regions are not handled in most models, but arise naturally in applications in vehicle control discussed in Chapter 6.

Constraints in IvP problems are represented by individual pieces or groups of pieces within an IvP function that contain a negative-infinity value as an interior function. In practice, some value, M, that is sufficiently large, depending on the number of objective functions, will suffice to represent infinity. The result is that constraints hold no special status in IvP functions, the implied feasible space plays no explicit role in the solution process, and the use of IvP is in no way dependent on the nature of constraints in any potential application. This is not to say however that "constraints" that exclude large portions of the decision space have no impact on solution speed in IvP problems.

3.2.4 Loosely Coupled IvP Problems

In the general interval programming problem, there are k objective functions defined over n decision variables. The decision space is the Cartesian product of the domains of each of the n decision variables. This number of dimensions has a strong impact on the solution process described in Chapter 5. In many situations however, the individual objective functions are defined over a small subset of the collective set of decision variables. We call such problems *loosely coupled* IvP problems.

In figure 3.8 the idea is illustrated with the graph theory analogy of dense and sparse graphs. A node is associated with an objective function, and an edge is drawn if the two functions are defined over a common decision variable. The tightly coupled problems (on the left) provide the greatest

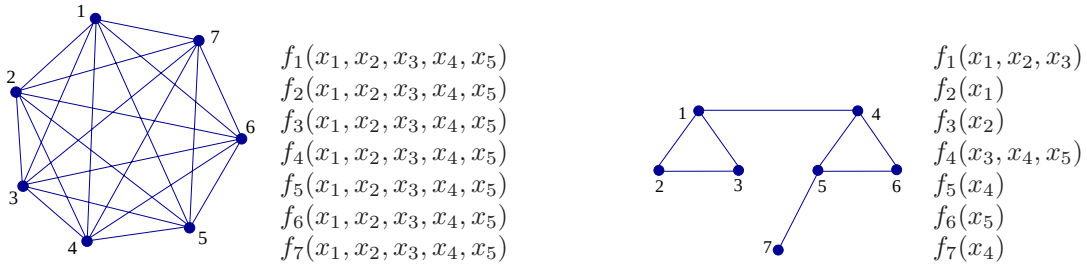


Figure 3.8: Tightly coupled vs. loosely coupled set of functions.

challenge. In the other extreme (not shown), where the corresponding graph is disconnected, we would have two independent, simpler optimization problems that could be solved separately. In between, there exist loosely coupled problems (on the right) where significant opportunities exist for capitalizing on these circumstances. Methods to exploit these opportunities will be discussed in Chapter 5.

3.3 Strengths of the Interval Programming Model

The strength of the IvP model is its mix of accuracy, speed, and flexibility. By flexibility, we mean its ability to be applied to a wide set of problems with few restrictions. In this section, we put IvP into the context of three other classes of methods that each have strengths in two but not all three of these measures, as shown in Figure 3.9. We will discuss these classes in Sections 3.3.1 thru 3.3.3, and argue that IvP takes a bit of the positive qualities, while avoiding the major pitfalls of each.

3.3.1 Avoiding the Speed Shortcomings of Full Brute Force Methods

One of the simplest ways to search through a decision space for an optimal decision is to just evaluate *all* possible decisions with respect to each objective function. This algorithm is perfectly *accurate*, and guaranteed to terminate after a finite number of steps, given the assumption of a uniform discrete

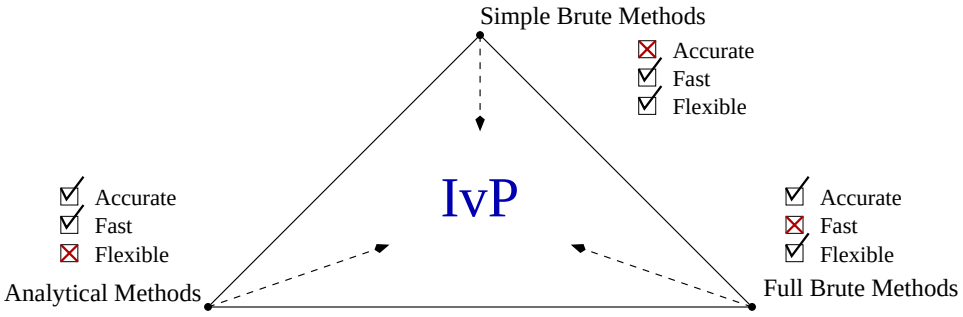
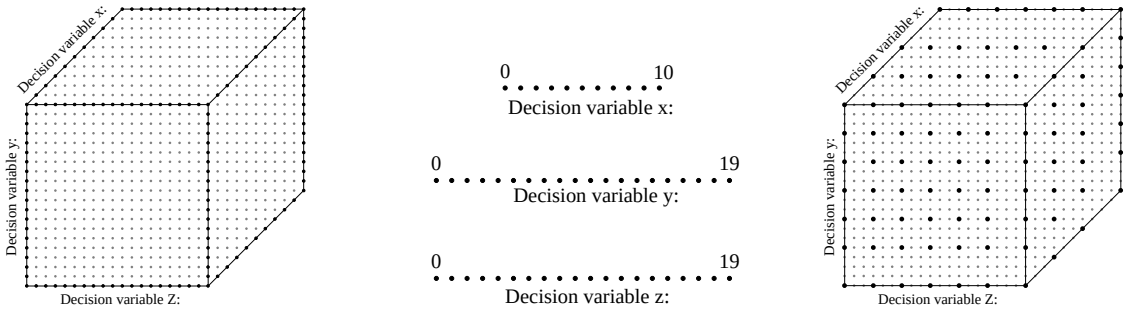


Figure 3.9: IvP is fast, flexible and accurate by taking the best of three types of methods.

domain (Section 3.1.1). This approach is also extremely *flexible*, since the only requirements are the ability to iterate through the decision space, and that each objective function be able to evaluate any point in the space. In fact, the objective functions themselves can also be defined in a brute force manner, explicitly pairing each point in the decision space with a numerical value. This is the approach taken in Rosenblatt (1997) for example. The drawback of course is that the size of such a decision space grows exponentially with respect to the number of dimensions. This typically leads to the need to "simplify" the decision space as discussed below in Section 3.3.2. As Rosenblatt (1997, p. 142) states: "the combinatorics require prohibitively expensive computations on the part of all behaviors, as well as the arbiter; this is exacerbated as the dimensionality or the resolution within any single dimension increases." The IvP model avoids these shortcomings by avoiding any exhaustive enumeration or iteration through the decision space.

3.3.2 Avoiding the Accuracy Pitfalls of Simplified Brute Force Methods

There are two common ways to counteract a decision space that has become unmanageably large. If the decision space contains n dimensions, then the first way is to reduce the single nD decision into a sequence of n decisions in 1D. This is depicted in Figure 3.10(b) below where the 3D decision space containing $20 * 20 * 11 = 4400$ points, is reduced to three separate decisions, each with a small number of choices. The drawbacks of this technique were discussed earlier in Section 2.2.4. Typically the setting of the first variable influences or constrains the setting of later variables. There are many variations on this approach ranging from dropping one or more of the variables completely (equivalent to presetting it to a fixed value), or separating, say, a 6D decision into two 3D decisions rather than six 1D decisions.

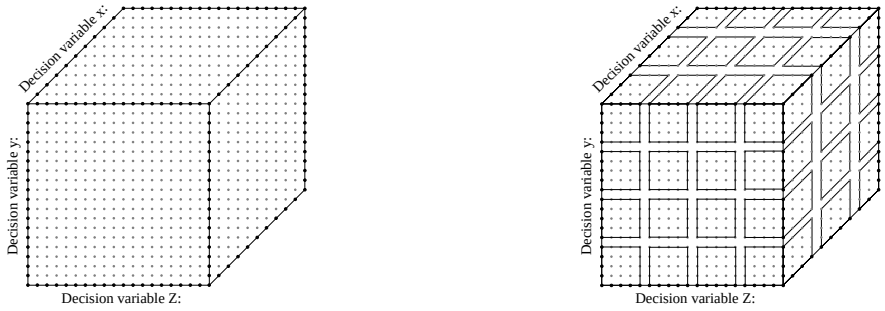


(a) Full decision space (b) Three separate 1D decisions (c) Reduced decision space

Figure 3.10: Two poor ways, (b) and (c), to counter-act a large decision space (a).

The second common way to counter-act a large decision space is to reduce the resolution of each decision variable, as depicted in Figure 3.10(c). Since the "proper" resolution is a rather subjective design decision, it is more difficult to declare that such a compromise is in effect. It is also less effective in reducing the decision space, since the space still grows exponentially. Both of these simplification approaches result in potentially severe compromises in accuracy.

The IvP approach offers another alternative that does not suffer from these two pitfalls. In an IvP function all points are contained in one of the pieces in a IvP function, as depicted in Figure 3.11(b). The number of pieces is typically much smaller than the number of points in the decision



(a) Full decision space (b) IvP piecewise function

Figure 3.11: Using a piecewise representation, (b), to counter-act to a large decision space (a).

space, with each piece implicitly defining the value of each interior point with its interior function. It is worth noting that, when this interior function is a constant rather than linear function, the situations in Figures 3.10(c) and 3.11(b) are nearly similar. By reducing the variable resolution as in Figure 3.10(c), in effect, one is mapping all the dropped neighbor points (the lighter points in the figure) to have the same value as the closest non-dropped points (the darker ones in the figure). In a piecewise constant function, one is also mapping all interior points to a single value; typically the value of the center point. We say "nearly similar" above since the piecewise constant IvP function

is not bound to be uniform, as is the resolution reduction approach. Later, in Section 4.3.3, we compare the accuracy of uniform piecewise constant functions against uniform piecewise linear and non-uniform piecewise linear functions. We assume that the accuracy performance present in the variable resolution reduction approach is comparable to the performance in the uniform piecewise constant functions.

3.3.3 Avoiding the Flexibility Pitfalls in Analytical Methods

Many analytical methods offer a potentially better solution to the problem of unmanageably large decision spaces. If, for example, one is able to cast each objective function into a linear, or quadratic function, the size of the decision space becomes largely irrelevant. The pitfall is that many problems of interest to us here, are not easily cast in such convenient forms. An approach that strictly limits the *form* of the underlying objective function is likely to disqualify itself from being of any use in applications of interest here. Note that the brute force methods discussed in Sections 3.3.1 and 3.3.2 above, have no such restrictions. In Chapter 6, the objective functions typically found in vechicle motion planning are discussed, and we will see that indeed, very little can be assumed about the form of these objective functions.

Chapter 4

Creating Interval Programming Functions and Problems

The purpose of this chapter is to demonstrate how IvP functions and IvP problems are created, and to show that they can be created with sufficient speed and accuracy to be useful in the target applications and beyond. We recognize that there is a significant existing body of methods for approximating functions with piecewise linear functions and their variations. We don't claim to present algorithms here that represent any improvement whatsoever. However, we did need to implement *some* algorithm for generating IvP functions in order to proceed with our methods for combining them as multiple objective functions. Our naive methods, presented in this chapter, were more than sufficient for building IvP functions in our simulator, discussed in Chapters 6 and 7. Furthermore, certain issues presented in this chapter are likely to be relevant regardless of how sophisticated the algorithm.

4.1 Making IvP Functions From Non-IvP Functions

Recall that a piecewise defined IvP function typically is an approximation of some underlying function, and typically is generated through some automation process. In creating IvP functions, three competing objectives are present. We want the formulated problem to be accurate, as well as not time-consuming to create, and not time-consuming to solve. The relationship between these three objectives is played out primarily in the number of pieces in each function, as well as the effort spent in choosing better piecewise configurations. The choices of piece shape and interior function are also important, but less contentious.

Building IvP functions is the first of the two most computationally expensive parts of the control loop shown in Figure 4.1. This stage is unique to the IvP approach and may seem counter-intuitive at first since we are adding inaccuracies, as well as time, to a process in which we primarily seek to

minimize loss of accuracy and time. The justification for this, addressed in Section 3.3, is the need

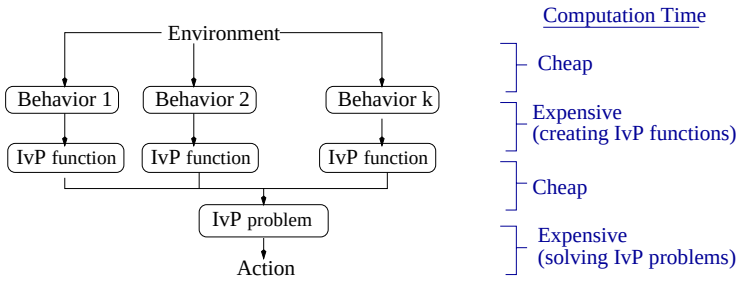


Figure 4.1: Components of the basic control loop and relative computation loads.

for an overall method that is fast, accurate, and flexible by making small sacrifices in each to achieve the right mix of the three. In Section 4.1.1 the relationship between the choices and performance in these two key stages are discussed. In Section 4.1.2 the algorithms for creating IvP functions are presented, and Section 4.1.3 issues concerning higher dimensional problems are discussed.

4.1.1 Time, Resources, and the Capacity for Accuracy

In a piecewise defined function, the number and shape of pieces, and their interior functions are *resources* that determine the *capacity* for accurately expressing an underlying function. These resources also affect the solution time when combining this particular function with others in an IvP problem (Chapter 5). The process of choosing a particular configuration of pieces also can vary in time and effort, and directly affects the degree to which the capacity for accuracy of the function is realized. The use of more resources means the ability to achieve a greater accuracy quicker. In solving IvP problems, the use of greater resources, translates into longer solution times. As will be shown in Section 4.3.1, the capacity for accuracy is bounded by the resources available, and the quality of the algorithms. These algorithms are discussed next.

4.1.2 The Basic Algorithm

The basic idea of the algorithm is to start with a piecewise defined function, where the set of pieces, S, initially contains only one piece (the entire decision space) and some arbitrary interior function, and then repeat the following loop until we are satisfied:

```
while(satisfied == FALSE)
    p_i = choosePiece(S)
    p_j = splitPiece(p_i)
    refinePieces(p_i, p_j)
    S = S ∪p_j
```

Figure 4.2: The general algorithm for building an IvP function.

The four important questions are then 1) How is the piece chosen? 2) How is the piece split? 3)

How are the two new pieces refined? 4) When should the algorithm stop? There are many different ways to implement these aspects. We present here some choices that we have implemented, and for which we will present empirical results in Section 4.3.

Selecting a piece The simplest way to select a piece is to pick one randomly. There are two ways to do this. First we could select a random element in the set S. When S is an array, this is very quick and simple. The second way to select a random piece is to generate a random point in the decision space, and then find out what piece it currently belongs to. This has the advantage that larger pieces (with typically more error) will have a greater chance to be selected for refinement. The drawback is that, given a point in the decision space, it is not trivial to determine which piece it corresponds to. For example, we would not want to search through an array representing S after selecting each point. We have made this issue less troublesome by keeping a grid structure corresponding to S.

To be more selective about piece selection, we could randomly choose pieces as above, but sample a piece first to see how well the piece currently reflects the underlying function. After examining a certain number of pieces, we then choose to proceed with the worst piece. The question here is how to evaluate a piece. The simplest way is to pick one random point inside the piece, and compare its evaluation based on the piece interior function against the underlying function, to give a measure of error. But one sample point may be misleading. Some number of sample points should be chosen that balances speed and accuracy. Furthermore, between iterations we would like to remember the bad pieces from the previous iteration that fell just short of being the worst. These are good candidates for choice in the current iteration before any more work is done. To implement this, we utilize a special "fixed-length" priority queue of prime candidates for refinement.

Splitting a piece The simplest way to split a piece is to split it in half along a randomly chosen dimension. Another way is to split it along the longest dimension, keeping its aspect ratio to a minimum as a rule of thumb. Of course, if there is some knowledge of the underlying function that can be capitalized on, this may allow for a more intelligent choice. Another choice is to try different splits and refinements, and choose the best configuration. In practice, we generally split along the longest dimension while randomly breaking ties.

Refining a piece To refine a piece, we choose a new interior function by selecting enough points in the piece, and based on their evaluation by the underlying function, choose the best parameters for the new interior function. In the case where the interior function is a constant value, we could simply choose the average of the selected points, or choose another number that minimizes the worst error. In the case of linear interior functions, the method we implemented is to first evaluate each corner of the piece (using the underlying function). Next we choose appropriate sets of $n+1$ points which determine a (hyper)plane in n dimensions, and simply average the coefficients of the linear function. The difficult question, especially in higher dimensions, is how many of these sets to sample, since the number of subsets grows exponentially. In practice, we sample $2n-1$ of the 2^n subsets.

Knowing when to halt The simplest criteria for halting is when some pre-allocated limit has been reached, either on the number of pieces, or time. A more difficult criteria may be based on the measured accuracy of the piecewise defined function in representing the underlying function. Unless an exhaustive evaluation of each point in the decision space is carried out, it is difficult to precisely assess the accuracy of the function. However, through sampling enough of the space, a picture of the situation can be drawn despite the lack of an absolute guarantee. In practice, we typically halt when a certain amount of pieces have been generated.

4.1.3 Issues in Higher Dimensions

As we will see in Section 4.3.2, the greater the number of dimensions in an underlying function, the lower the capacity for accuracy in a piecewise defined function of fixed resources. The relationship is roughly shown on the left in Figure 4.3. The primary concern is that such errors are additive when

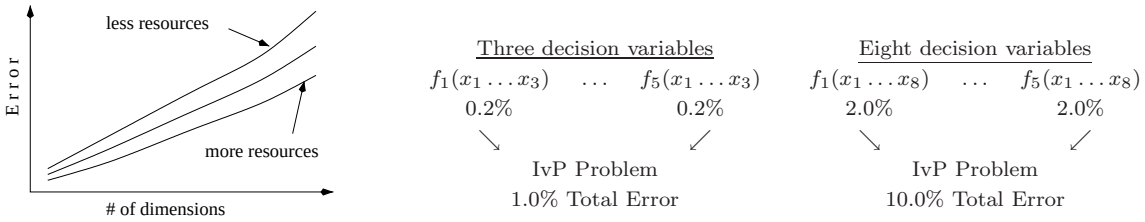


Figure 4.3: Capacity for accuracy and error propagation with respect to dimensions.

combining several objective functions in an IvP problem, as portrayed on the right in Figure 4.3. While the algorithms presented in Chapter 5 guarantee global optimality, given their input, errors introduced in building IvP functions erode the value of this guarantee.

There are two points to mention that soften the concern for this problem. The first is that, even with the errors introduced in building IvP functions, a bound can still be put on the cumulative error associated with the produced solution. This "global" optima comes then with the caveat that, if it's not the optimal solution, then the optimal one is better by at most this bounded amount. This situation is still preferable in most cases to the lack of any guarantee that comes with a purely local search method. The second point is that, in practice, the errors tend to cancel out each other. For any given point in the decision space, the piecewise defined approximation will either under- or over-value the point, with typically no particular regularity.

There are two special (but common) cases of IvP problems where the number of decision variables for the problem can be quite large, yet the problem with inaccuracies in piecewise defined functions is relatively small. The first case is found in loosely coupled problems, discussed in Section 3.2.4. The individual objective functions that are defined over a small subset of the entire set of decision variables, are likely to have a greater capacity for accuracy, resulting in overall less error accumulation despite the higher number of decision variables.

The second case is seen when the individual objective functions utilize non-rectilinear boundary functions, as discussed in Section 3.1.4. Suppose we have a problem with 100 decision variables, so each objective function will have an underlying function of the form $f(x_1, \ldots, x_{100})$. But suppose the underlying function is dependent on two properties, g, and h, which are functions over a subset of the variables, say $g(x_1, \ldots, x_{50})$ and $h(x_{50}, \ldots, x_{100})$ respectively. By utilizing pieces with intervals over the two functions, g and h, rather than over the variables x_1 thru x_{100}, we can expect the accuracy of the IvP function to be typical of a $2D$ function rather than $100D$ function.

4.2 Methods for Collecting Empirical Results

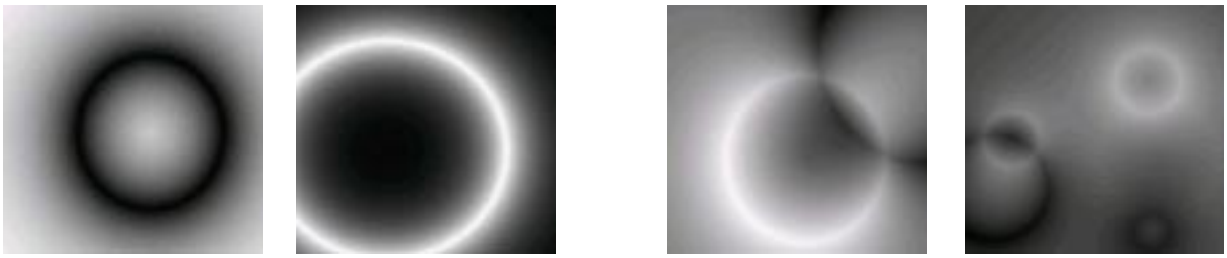
In this section, three preliminary items are addressed before diving into the empirical results of Section 4.3. They are: a) What kinds of functions to test (Section 4.2.1), b) How to measure effectiveness (Section 4.2.2), and c) How the reproducibility of test results is ensured (Section 4.2.3). The first of these issues is the most important and contentious. It is recognized that the usefulness of IvP cannot be measured by how well it handles artificially contrived problems. But on the other hand, it can not be measured by how well it impacts a single "real world" problem. The strategy here is to make a convincing case by doing both.

Later, in Chapter 6, IvP will be used to model navigation decision making on an autonomous underwater vehicle, with competing considerations between path following and avoiding other moving vessels. The series of IvP problems in this application will have more-or-less consistent characteristics from one decision to the next. One may then wonder how IvP performance (either accuracy or speed) would change if the application required different characteristics, such as different decision variables, a tighter control loop, or more moving contacts.

The empirical results presented at the end of this chapter work on contrived problems not likely to be found in practice, but will shed some light on how the variation of certain characteristics tend to impact speed and accuracy performance. We aim to learn as much as possible by focusing on a particular meta-class of functions, described below in Section 4.2.1. We choose this class of functions because: a) we expect their properties, such as non-linearity and non-convexity, to pose at least as great, or greater difficulties as those we expect to find in our target applications, and b) they are easy to generate by using random number generators to set parameters, giving a function with very different properties with a slight variation in the right parameter.

4.2.1 Ring Functions

A *ring function* is built by placing a $n-$dimensional ring somewhere in $n-$space, by choosing both its center and radius, and then associating values to all points in the domain based on the distance to the ring. Simple examples are depicted below in Figure 4.4(a) and (b). More than one ring can be used as shown on the right in Figure 4.4(c) and (d). We can see that, even with one ring, we are dealing with a non-linear, non-convex function. As more rings are added, the number of local optima



(a) (b) (c) (d)

Figure 4.4: Example ring functions with one (a), (b), two (c) and five (d) rings.

also tends to rise. From the definition given below, we will identify the parameters of interest.

A *ring function* is a function with the following form:

$$f(x, y) = f_1(x, y) + \ldots + f_p(x, y),$$

where each of the p functions $f_i(x, y)$ correspond to a particular ring given by:

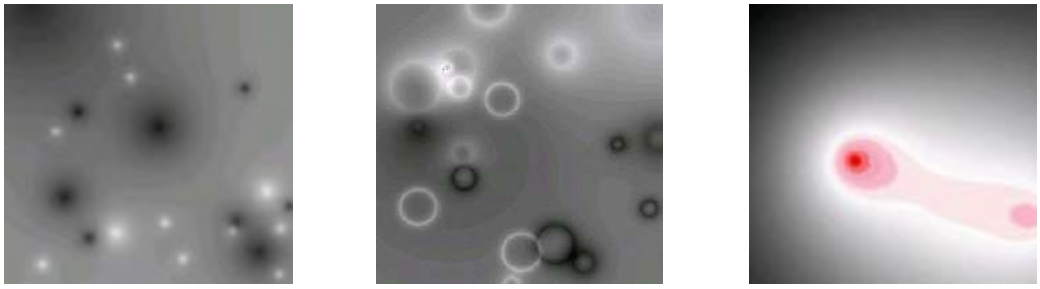
$$f_i(x, y) = ((1 - \frac{|\sqrt{(x-h)^2 + (y-k)^2} - \texttt{rad}|}{\texttt{max-dist}})^{\texttt{exp}} * \texttt{range}) + \texttt{base}$$

where each function $f_i(x, y)$ is equal to $g(r(x, y))$ given by:

$$r(x, y) = |\sqrt{(x-h)^2 + (y-k)^2} - \texttt{rad}|, \qquad g(x) = ((1 - \frac{x}{\texttt{max-dist}})^{\texttt{exp}} * \texttt{range}) + \texttt{base}$$

The function $r(x, y)$ indicates a circle (ring) with radius `rad`, and returns the shortest distance of a point, (x, y) to the ring. The function $g(x)$ takes this distance and produces the desired value based on the following intuition. The center of each ring is set to be somewhere in the universe given by the Cartesian product of each variable's domain. The value of `max-dist` is the maximum distance in this universe, i.e., the length from corner to opposite corner of the universe. The value of $\frac{r(x,y)}{\texttt{max-dist}}$ is therefore always in the range $[0, 1]$. Subtracting $\frac{r(x,y)}{\texttt{max-dist}}$ from 1 and raising it to the exponent `exp` still leaves us with a value in the range $[0, 1]$. Multiplying this by `range` and adding it to `base`, ensures that each function, $f_i(x, y)$, is guaranteed to range over $[\texttt{base}, \texttt{base} + \texttt{range}]$, and thus $f(x, y)$ has the range $[p(\texttt{base}), p(\texttt{base} + \texttt{range})]$. The actual range of the function may be quite smaller and is unknown based solely on the parameters.

By controlling the parameters, p, `rad`, `exp`, `base`, and `range`, as well as the number of variables and their domains, we can get functions with rather predictable properties. For example, by using a large number of rings with a small radius and large `exp` factor, instances such as shown below in Figure 4.5(a) are typical. By then bumping up values to the radius parameter, instances such as that in Figure 4.5(b) are common. And finally by lowering the ring parameter, p, and the `exp` parameter, instances like that in Figure 4.5(c) are typical. By first setting the above parameters, by and then



(a) (b) (c)

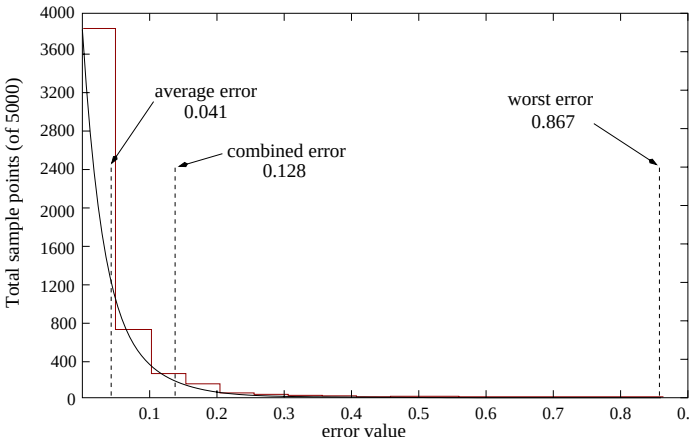
Figure 4.5: Three types of ring functions based on varying parameter inputs.

letting the ring positions vary randomly we can generate wildly different looking functions within a class of functions with certain expected properties. This is the situation we desire for testing the algorithms designed to build accurate piecewise defined functions. In Section 4.2.3 and Appendix B, more detail is given on how these functions are generated and stored for experiments.

4.2.2 Accuracy Measurement Issues

The accuracy of an IvP function, $f(x_1, \ldots, x_n)$ in representing another function $g(x_1, \ldots, x_n)$ is measured by selecting some or all of the points $\langle x_1, \ldots, x_n \rangle$ in the $n-$dimension domain, evaluating each using f and g, and tallying their (absolute) difference. Primarily we are concerned with how well the approximation performs on average (`avgErr`), as well as in the worst case (`worstErr`). In testing, we record both, but use another single number (`repErr`) that is a combination of the two, in most of the reporting. They are defined as follows: The average error, `avgErr`, over s sample points is: $1/s \sum_{i=1}^{s} |(f(\mathbf{x_i}) - g(\mathbf{x_i}))|$. The worst error, `worstErr`, is simply the worst error of all the sample points. The combined error, `repErr`, is a combination of the two given by: `avgErr` $*$ (0.1)`worstErr`.

In Figure 4.6, the relationship between these three values is depicted. The data for this graph was obtained by creating a single IvP, piecewise linear function with 1000 pieces, approximating a particular 2D ring function (actually, one of the 2D functions described below in Section 4.3.1). After sampling 5000 points, the error values for each point are put into one of the intervals shown in the table. The value plotted on the y-axis is the percentage of total points contained in the interval. The error value for a particular sample point is normalized over the range of the function, to be comparable to a function with a range of 100. For example, if the average error is 0.80, and the function ranges from -150.00 to +250.00, the normalized average error is: $0.80 * \frac{100}{250-(-150.00)} = 0.20$. Of course we may not precisely know the range of the function unless we sample the whole domain. Barring this, the range is also approximated by taking the min and max after sampling. Within a particular experiment, the number of sample points is typically fixed, and disclosed.



Error range	# pts	% of total
[0.00, 0.50]	3714	.7428
(0.05, 0.10]	756	.1512
(0.10, 0.15]	287	.0574
(0.15, 0.20]	133	.0266
(0.20, 0.25]	41	.0082
(0.25, 0.30]	27	.0054
(0.30, 0.35]	17	.0034
(0.35, 0.40]	10	.0020
(0.40, 0.45]	2	.0004
(0.45, 0.50]	5	.0010
(0.50, 0.55]	3	.0006
(0.55, 0.60]	1	.0002
(0.60, 0.65]	2	.0004
(0.75, 0.80]	1	.0002
(0.85, 0.90]	1	.0002

Figure 4.6: The average, worst, and combined error for a single experimement sampling 5000 points.

4.2.3 Reproducing the Results

The experiments reported on in Section 4.3 below are each based on a set of ring functions. It is intended that these functions be accessible to facilitate the improvement and verification of solution algorithms. The ring functions are stored in a formatted file called a ".rng" file, detailed in Appendix A. The creation of an IvP function is stored in a ".ipf" file also detailed in the same appendix. An IvP function is created from a ring function by the invocation of the `solveIPF` executable. The parameters for this function are also described in Appendix A and correlate to the basic algorithm description provided previously in Section 4.1.2. The particular `solveIPF` parameters used in an experiment, and the names and locations of .rng and .ipf files are given in Appendix B.

4.3 Empirical Results

The results in this section focus on how well we can approximate a given function with a piecewise defined IvP function. In particular, we would like to measure accuracy as a function of time, and the various parameters discussed in Sections 4.1.1 and 4.1.2.

4.3.1 Accuracy vs. Resources vs. Time

The objective of this experiment is to demonstrate the relationship between resources, time and accuracy. Recall that in creating an IvP function, there are three important objectives: 1) be as accurate as possible, 2) use as few pieces as possible, and 3) be as quick as possible. In different applications, the priority of each will be different.

In this experiment, we create a group of 4D ring functions detailed in Appendix E.2, and we create IvP functions using 1K, 2K, 5K, 10K, and 20K pieces. Within each of these five groups we

vary the amount of "effort" spent in selecting the best piece to refine, in the select-split-refine loop described in Section 4.1.2. In this loop, effort could be varied in each of the three parts, each likely improving the end product. In this experiment, we chose to gradually modify the effort spent in the first part, through the `-queSamp` parameter, which determines how many points are sampled and put into a fixed length priority queue on each iteration. (This parameter and other parameters to the `solveIPF` executable are described in more detail in Appendix A.) The results are shown below in Figure 4.7. Each of the five groups/lines shown represent a run on the same 50 ring functions

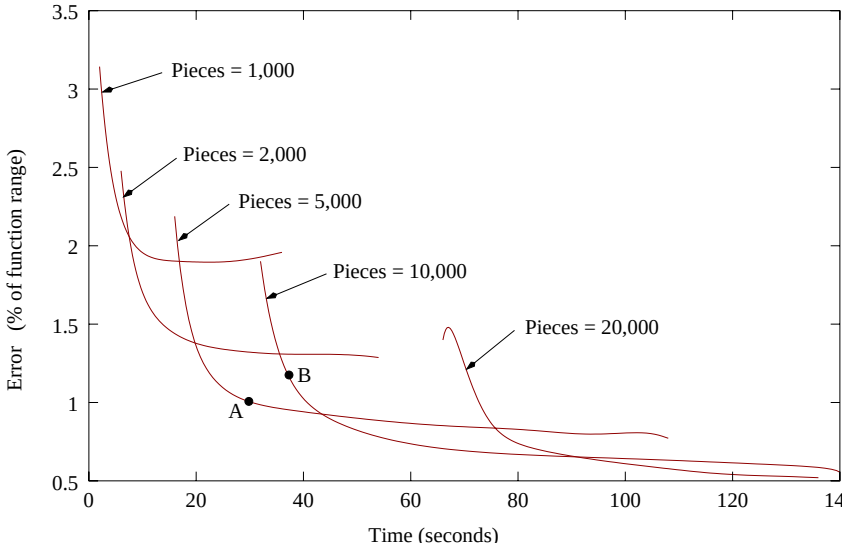


Figure 4.7: Accuracy vs. time while varying piece count and number of sample points.

describe above, with 25 different variations in the value of the parameter `-queSamp` ranging from 1 to 700. Each line represents roughly 1200 test runs.

Observation 1: As the number of pieces is increased, the capacity for accuracy is also increased, but the effort/time needed to generate the IvP function is also greater. When considering strictly the *capacity* for accuracy, the time it takes to reach this capacity, and the piece count, then there are no dominating points in this space. In other words, an improvement in one must come at the expense of one of the other two, or both.

Observation 2: When considering individual points, such as A and B in Figure 4.7, there are indeed dominating points. For example, point A not only uses fewer pieces than B, but is also faster to generate and more accurate. The fact that having 5000 pieces at one's disposal, as in A, translates into a lesser *capacity* for accuracy than having 10000 pieces, as in B, is irrelevant in practice. The trick for the practitioner is to find this elbow in the curve and use it to their advantage.

4.3.2 Piecewise Linear vs. Piecewise Constant IvP Functions

In this experiment, the aim is to compare the effectiveness of piecewise linear vs. piecewise constant functions, described in Section 3.1.3. It is expected that piecewise linear functions are preferable for their greater modelling capacity, but they are also more time-consuming to create, and to solve, when combining a group of them in an IvP problem instance.

In this experiment, we create 9 groups of ring functions ranging in dimension from 2D to 20D, (where each decision variable corresponds to one dimension). Each group has 10 randomly generated instances. See Appendix E.3 for the details on these functions. For each of these 90 ring functions, we created 6 IvP functions. Two with 3000 pieces, two with 8000, and two with 15000. In each pair, one is created piecewise linear, and the other piecewise constant. The results are shown below in Figure 4.8.

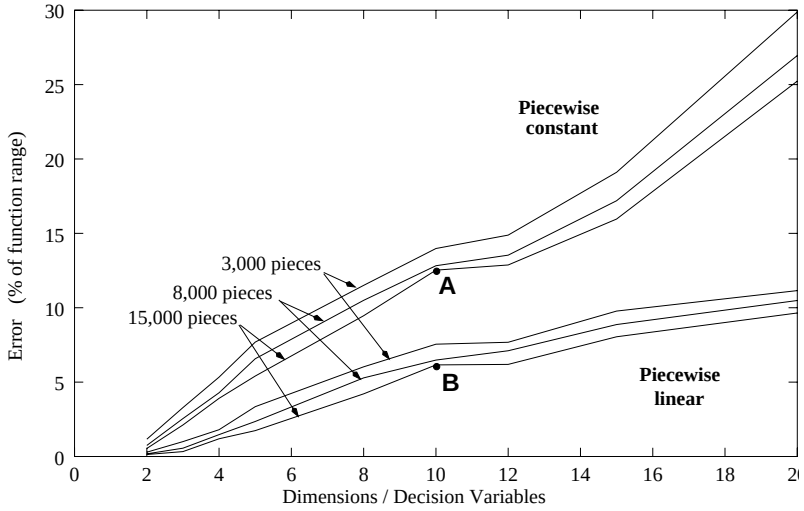


Figure 4.8: Accuracy vs. number of dimensions for piecewise linear and piecewise constant functions.

From these results, we confirm the piecewise linear functions are indeed more accurate, with an increasing gap in higher dimensions. But consider the two points, A and B, in Figure 4.8. They both represent the error in making an IvP function from the same set of 10 ring functions. Point A represents the piecewise constant, and point B the piecwise linear results. Point B appears to be the clear winner except that, on average, these IvP functions take 296.6 seconds to create, compared to the 25.8 seconds needed to create the piecewise constant IvP functions. (Creation times for all points can be found in Table E.4, Appendix B.)

It appears that we are in a tradeoff situation, like the one described previously in Section 4.3.1, where one must weigh the need for being accurate against the need for being quick. But if we take the same set of ring functions and pick a dimension, say 8D, and plot the error as the piece count grows, we see a trend like the one shown below in Figure 4.9. The two points in Figure 4.9 represent one IvP function that is piecewise linear with 100 pieces (point A), and one that is piecewise constant

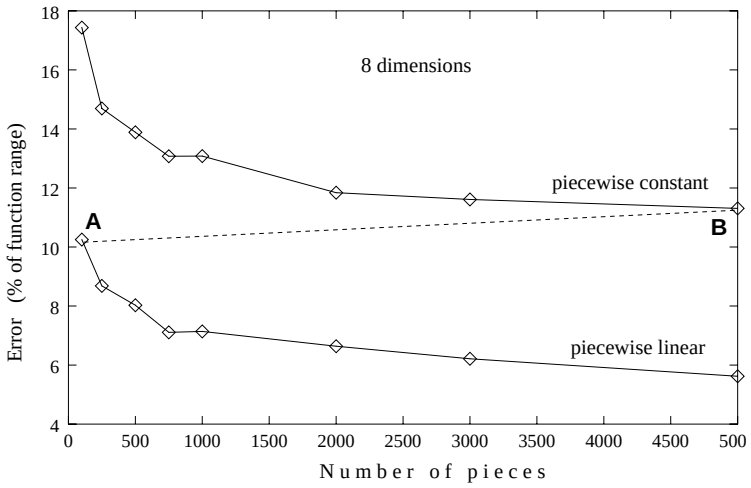


Figure 4.9: Error vs. piece count in 8D for piecewise linear and piecewise constant functions.

with 5000 pieces (point B). The first function not only takes less time to make, but is also more accurate. Furthermore IvP piecewise linear *problems* with 100 pieces can be solved quicker than piecewise constant problems with 5000 pieces. If the piece counts were identical, then indeed, the piecewise constant problems could be solved more quickly, for reasons explained later in Chapter 5. But the gross difference in piece count, in this case, more than compensates. Our conclusion is that piecewise linear functions should be preferred in virtually every situation. (A reasonable question then is: why stop at piecewise linear? Why not piecewise quadratic etc.?)

4.3.3 Uniform vs. Non-uniform Pieces

In this experiment, the aim is to compare the effectiveness of uniform vs. non-uniform pieces. The distinction between a uniform discrete decision space, and uniform pieces, was made in Section 3.1.2. Here we compare the following three types of functions:

- Uniform piecewise constant functions.
- Uniform piecewise linear functions.
- Non-uniform piecewise linear functions.

From the previous section (4.3.2), we know that piecewise constant functions are dominated by piecewise linear functions with respect to capacity for accuracy and creation time. We include them in this experiment because in Section 3.3.2 we claimed that uniform piecwise constant IvP functions have similar characteristics, with respect to capacity for accuracy, to the variable resolution reduction methods mentioned in that section. For this experiment, we generated 120 different ring functions, 60 in 3D, and 60 in 2D. Within each of these groups of 60, we varied the `-expRng` parameter, which determines the slope or "intensity" of a rise in a particular peak. The data from all six groups is reported in Appendix E.4, and the data for one of the groups in 3D is depicted below in Figure 4.10.

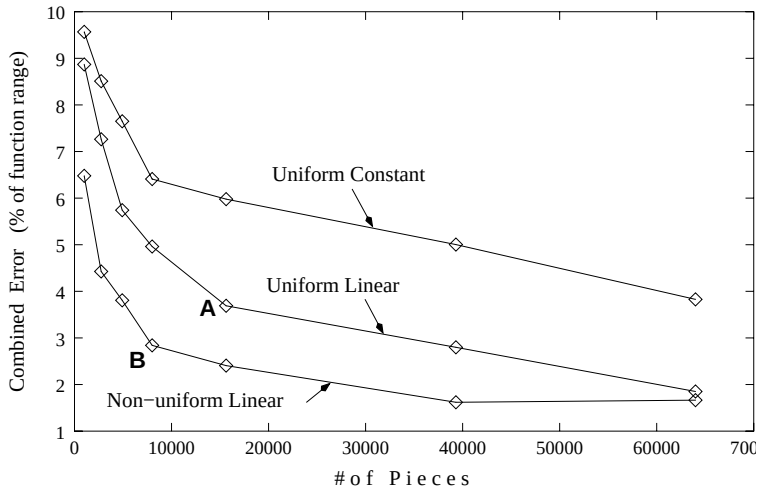


Figure 4.10: Accuracy vs. piece count for uniform-constant, uniform-linear and non-uniform-linear IvP functions.

From this data we see that there is a clear trend in the capacity for accuracy between the three types of functions. However, consider points A and B in Figure 4.10. The combined error for functions at point B is 2.84 vs. 3.69 at point A. But the average creation time for point B is 39.14 seconds vs. 33.53 for point A. Likewise comparisons between other points in Table E.6 and Table E.9 in Appendix B reveal mixed results. In some cases, better accuracy can be achieved with the non-uniform pieces, in less time, *and* using fewer pieces. In other cases, a tradeoff is apparent.

At this point, the jury is still out on whether the non-uniform pieces are *always* worth the extra time for the sake of extra accuracy. However, three points can be noted. First is that methods used here for generating non-uniform piecewise functions have a lot of room for improvement, whereas the results for uniform pieces will not likely change. Second, hybrid methods hold much promise, such as covering the space initially with a set of uniform pieces, and then dedicating an additional number in a non-uniform manner. Such "semi-uniform" IvP functions will be addressed again in Section 5.2.5. Lastly, the whole point of allowing non-uniform pieces is that different objective functions, from different behaviors/goals, will likely need to dedicate pieces in different ways. In loosely coupled problems (Section 3.2.4), for example, some functions are defined over only a subset of the variables. Certainly, uniformity across these functions in loosely coupled problems is a waste of resources. Practically speaking, non-uniformity is an absolutely essential feature, but some work needs to be done on general methods to make it a clear winner in a wider variety of situations.

Chapter 5

Solving Interval Programming Problems

The solution to an IvP problem is a point in the decision space with the optimal combined value from each IvP objective function. To find this solution, an algorithm with the overall structure of branch and bound is used. In this chapter the Interval Programming ALgorithm (`IPAL`) is developed in Sections 5.1 thru 5.3. Two alternative solution methods are given in Sections 5.4 and 5.5. Finally, in Section 5.6, empirical results are given, comparing methods and performance as certain problem parameters vary.

5.1 Search Through the Combination Space

The search space in `IPAL` is the set of all possible combinations of m pieces from each of the k objective functions, which we refer to as the *combination space*. Given that there are m pieces in each function, the size of this space is m^k. How big is this space? Consider a simple example problem with 5 decision variables, each with 200 elements in its domain of $[0, 199]$. The size of the *decision space* is $200^5 = 6.4 \times 10^{11}$. Now suppose there are eight IvP functions with 100,000 pieces each, which makes the average piece have the size $20 \times \ldots \times 20$. The size of the *combination* space is 1×10^{40}. Why choose a search space that could be significantly larger than the decision space? The answer is that, even though situations can be contrived where every element in this space is a legitimate candidate, typically the vast majority of combinations are not legitimate candidates. By *legitimate* we mean combinations of pieces that all intersect, i.e., share at least one point in the decision space.

5.1.1 Intersection of Two Rectilinear IvP Pieces

The intersection of two pieces from different objective functions is a core operation in IvP algorithms. The intersection of two rectangles is simply the rectangular region of overlap, if they indeed overlap. In Figure 5.1, the intersection of two 2D rectilinear pieces is depicted algorithmically and graphically.

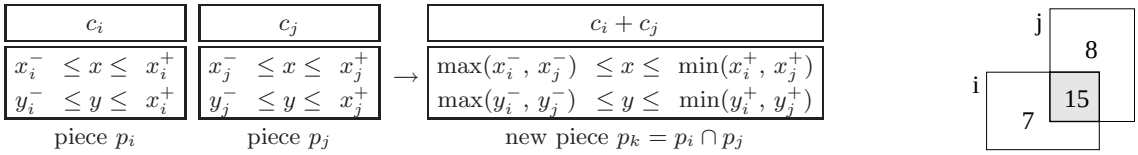


Figure 5.1: Intersecting two 2D rectilinear pieces.

A piece is *non-empty* if, for each of its intervals $v^- \leq v \leq v^+$, it holds that $v^- \leq v^+$. The boolean value of "$p_i \cap p_j$" is true if the resulting p_k is non-empty. For pieces with linear interiors, the only difference is the addition of the two linear interior functions, replacing "$c_i + c_j$" in Figure 5.1.

5.1.2 The Combination Space and Full Search Tree

This combination space corresponds to the tree structure shown below in Figure 5.2, where each *leaf* node corresponds to a combination of $k = 3$ pieces. Each node in the tree, except for the root node, corresponds to a single piece in some IvP function, referred to as the `nodePiece`. Each node also has

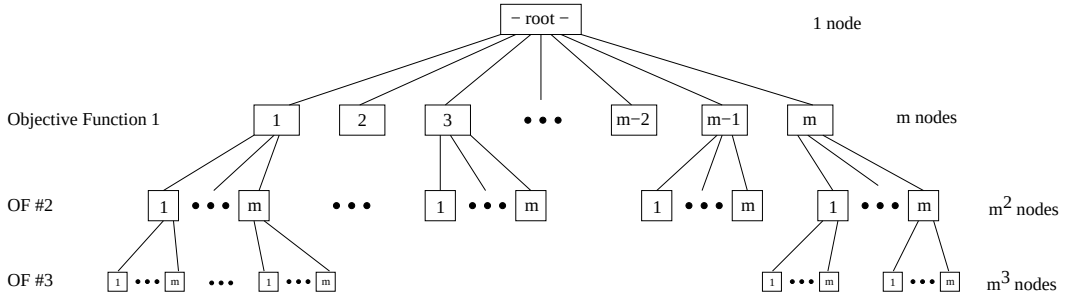


Figure 5.2: The Search tree for $k = 3$ objective functions with m pieces each.

a `nodeBox`, which is the intersection of its `nodePiece` and its parent's `nodeBox`. The values of points inside a `nodeBox` are given by the sum of the interior functions of each contributing `nodePiece` up the tree. The expression `nodeBox→wt` refers to the maximum weight of all such points, based on this combined function. (The `nodeBox` of the root is the universe, i.e., whole decision space, and the `nodeBox` and `nodePiece` for nodes at the first level are equivalent, i.e., contain the same points in the decision space.)

The algorithm below in Figure 5.3 traverses blindly through the entire combination space and will expand the entire search tree. The main function, `IPAL`, sets up the root node and launches the recursive call to `RIPAL` down the tree. The `nodeBox` at the root node is the entire decision space, given by the globally accessible value `universe`. The variables `bestBox`, `best`, and `uninitialized`

are also, for simplicity here, assumed to be globally accessible. The value, `best`, indicates the value of the maximum point within the working solution, `bestBox`. No initial best value is given, so when `uninitialized`, is true, the first leaf node will become the working solution, regardless of value. The first part of `RIPAL` (lines 0-6) checks for the boundary (i.e. leaf node) condition and potentially

```
IPAL()
0. uninitialized = TRUE
1. nodeBox = universe
2. RIPAL(nodeBox, 1)
3. return(bestBox)
```

```
RIPAL(nodeBox, level)
0. if(level == k)
1.    if(nodeBox is non-empty)
2.       if(uninitialized or nodeBox→wt > best)
3.           best = nodeBox→wt
4.           bestBox = nodeBox
5.           uninitialized = FALSE
6.    return
7. for(i = 1 to m)
8.    newNodeBox = nodeBox ∩ p(i, level)
9.    RIPAL(newNodeBox, level+1)
```

Figure 5.3: General structure of Recursive `IPAL`.

updates the best solution. The second part (lines 7-9) iterates through the pieces associated with that level and builds a new node to branch on. The expression `p(i, level)` denotes the ith of m pieces in the objective function indicated by `level`. This algorithm performs no pruning whatsoever and will expand all p^m leaf nodes, thus ensuring global optimality.

The aim is to evolve this algorithm, preserving the guarantee of global optimality, so that it prunes as much of the tree as possible. In doing so, the time saved by pruning is balanced against the time needed to recognize a valid pruning opportunity. The first pruning strategy comes from slightly altering the version of `RIPAL` in Figure 5.3 so that the check for an empty `nodeBox` is done at internal nodes rather than exclusively at leaf nodes. The revised algorithm is given in Figure 5.4.

```
RIPAL(nodeBox, level)
0. if(level == k)
1.    if(uninitialized or nodeBox→wt > best)
2.       best = nodeBox→wt
3.       bestBox = nodeBox
4.       uninitialized = FALSE
5.    return
6. for(i = 1 to m)
7.    newNodeBox = nodeBox ∩ p(i, level)
8.    if(newNodeBox is non-empty)
9.       RIPAL(newNodeBox, level+1)
```

Figure 5.4: A bit smarter (and great deal more effective) version of `RIPAL`.

The simple observation is that if a node has an empty `nodeBox`, none of its leaf children are

legitimate solutions. The check for non-emptiness on line 8 replaces the same check done on line 1 in the former version of RIPAL in Figure 5.3. In practice, this results in an enormous amount of pruning at very little cost. Not all pruning strategies are such clear winners. Broadly speaking, two types of pruning are possible: a) pruning illegitimate, i.e., empty nodes as done in the revised algorithm just discussed, and b) pruning legitimate nodes by bounding. Pruning strategies discussed after this point all capitalize on a special grid structure introduced below in Section 5.2. Once this has been defined, we return to pruning, and better versions of `IPAL` and `RIPAL` in Section 5.3.

5.2 The Use of Grid Structures in IvP Solution Algorithms

Grid structures are used in `IPAL` to facilitate the retrieval of two types of information at each (non-root, non-leaf) node in the search tree. The situation is depicted in Figure 5.5, where an arbitrary node has m children, and many leaf children.

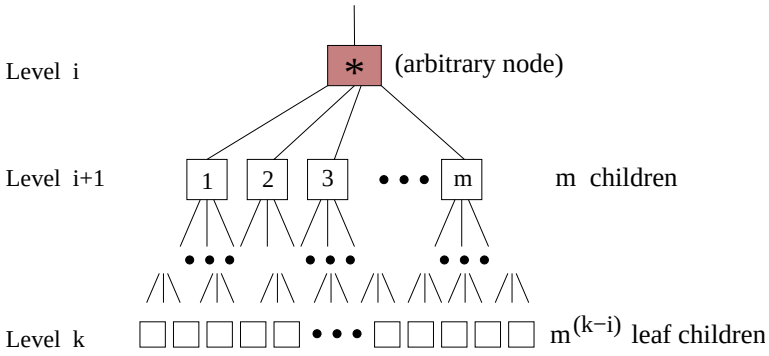


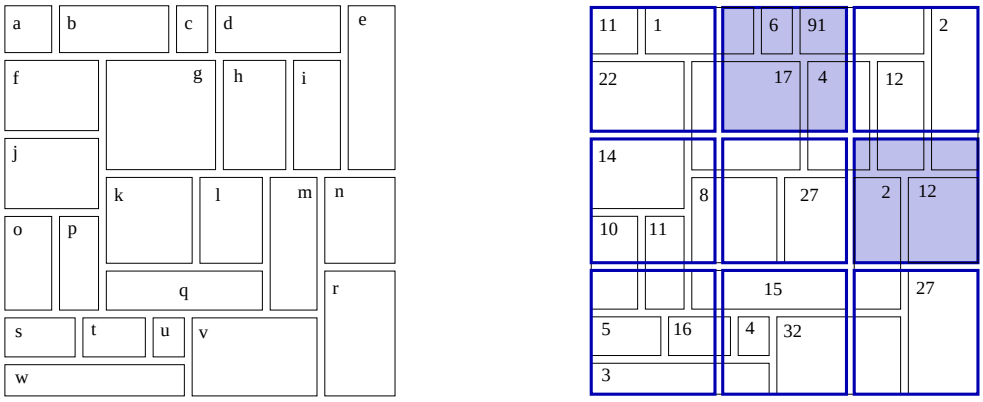
Figure 5.5: For a given node: Which children intersect? Upper bound on best leaf?

Any one of the m children can be pruned because, either it doesn't intersect the `nodeBox` associated with its parent, or all of its associated leaf children are hopelessly inferior to a previously-found solution. It is of course possible to simply check each of the m children to see if they intersect with the parent box, but, in practice, the great majority do not intersect. For the ones that are not even close, we would like to avoid the check altogether. The use of grid structures described here can drastically reduce these m checks at such nodes. Without them, even with an IvP problem with only two objective functions, our algorithm would be $O(m^2)$, with typical runs consistently at worst case performance. Furthermore, the grids can be used to quickly obtain an upper bound on solution for a particular node before any of the m children are tested for intersection. The expression `grid[i]` refers to the grid associated with objective function i. In Section 5.2.1 the construction and format of a grid is given. In Sections 5.2.3 and 5.2.4, the use of the grid for intersection tests, and bounding are discussed. In Section 5.2.5, issues concerning the proper choices of grid parameters are addressed.

5.2.1 The Basic Grid Structure and Contained Information

Grids are composed of a multi-dimensional array of "grid elements", with two pieces of information associated with each element: `gridVal` and `gridList`. The latter is an initially empty linked list. When a piece from an objective function is added to a grid, the set of grid elements related to the piece is determined, and then the piece is added to the linked list of each such grid element. The value of `gridVal` indicates the maximum value across all pieces contained in the gridList. When a piece from an objective function is added to a grid element, the maximum point within the piece is compared to the current `gridVal`, and updated if greater. For piecewise linear functions, the maximum point within the piece is first determined (always a corner). No check, however, is made as to whether this point is actually inside the grid element.

In the example in Figure 5.6(a), there are 23 pieces in an example objective function, and 9 grid elements in the grid shown in Figure 5.6(b) with constant function interiors shown for each piece.



(a) Example 2D objective function with 23 pieces

(b) Grid containing the function pieces

Figure 5.6: Example grid with 9 grid elements and 23 rectilinear pieces.

The top shaded grid element in Figure 5.6(b), has the `gridList` containing pieces $\{b, c, d, g, h\}$, and the rightmost shaded grid element has the `gridList` $\{h, i, e, m, n\}$. Note that piece h belongs to both gridLists. The `gridVal` associated with the latter grid element would be 12, the maximum of $\{4, 12, 2, 2, 12\}$.

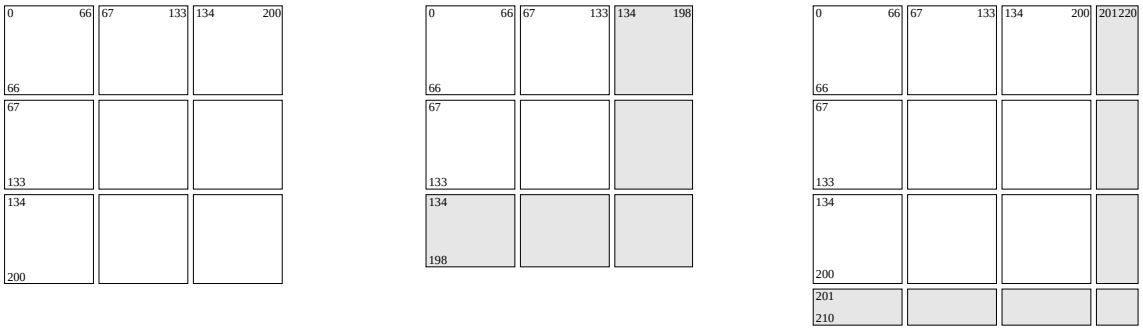
5.2.2 Initializing the Grid Element Layout

The initialization of the grid associated with objective function j takes two arguments as in:

`grid[j]→initialize(universe, gel)`

The `universe` is given by the Cartesian product of each variable's domain. It is equivalent to the domain of the corresponding objective function, and is also identical between objective functions

within an IvP problem. The second argument, `gel`, indicates the size of of each grid element. All grid elements within a grid will be the same size, provided that they divide evenly into the universe. For example, consider a 2D universe with both variable domains given by the interval $[0, 200]$, and a 2D `gel` with a length of 67 in both dimensions. The resulting grid is shown in Figure 5.7(a) with 9 identical grid elements. If the domains were instead $[0, 198]$, the higher indexed grid elements



(a) Uniform grid elements (b) Non-uniform grid elements (c) Non-uniform grid elements

Figure 5.7: Three different universes result in three different grid layouts with the same requested grid element size.

are truncated, resulting in a non-uniform grid, as shown in Figure 5.7(b). Likewise, if the variable domains were instead a bit larger, say $[0, 220]$, then additional grid elements would be created as shown in Figure 5.7(c).

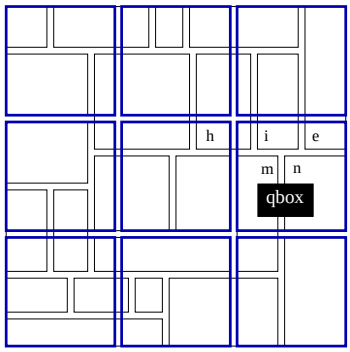
The uniformity of all but possibly the highest indexed grid elements is an important property that allows us to process a given IvP piece to directly determine which grid elements it corresponds to. Adherence to this property, for example, means that the grid elements in Figure 5.7(b) are as close to being identical as possible. In the example in Figure 5.7(c), the 16 grid elements could have been shaped in a more equitable way, which generally speaking, results in a more efficient grid. However, the initialization function does not override the requested `gel` size for the sake of such equity. The argument `gel` is typically related to the distribution of pieces in the underlying objective function. In practice, preserving this relationship is more important than improving the equity between the higher and lower indexed grid elements. The importance of this relationship is returned to in Section 5.2.5.

5.2.3 Retrieving Intersection Information

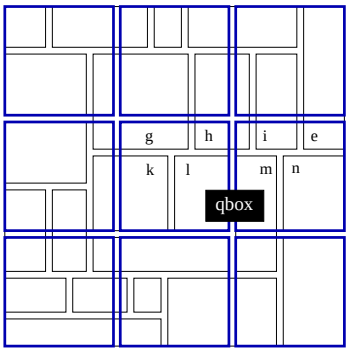
One important use of a grid is to query which boxes intersect with a provided *query box*. The syntax used in the algorithms provided later in Section 5.3 is:

- `boxset = grid[i]->getBoxes(qbox)`

where `grid[i]` is the grid associated with objective function i, `qbox` is a query box, and `boxset` is the resulting set (linked list) of boxes intersecting the query box. In the example depicted in Figure 5.8(a), the query box intersects one grid element containing boxes $\{h, i, e, m, n\}$. Individual pairwise intersection tests (Section 5.1.1) are performed on these five, with the returned list being $\{m, n\}$.



(a) Query box intersects one grid element



(b) Query box intersects two grid elements

Figure 5.8: Retrieving intersection information from a grid given a query box.

In the second case, in Figure 5.8(b), the query box intersects two grid elements. The concatenation of lists from these two grid elements is $\{g, h, k, l, h, i, e, m, n\}$, which contains piece h twice. This requires us to go through this list to remove duplicates. In practice this is done before the pairwise intersection tests are performed. In this simple case, the cost in removing duplicates is negligible, but in problems with higher dimensions (usually coarser grids) and more pieces, the removal of duplicates may consume a significant amount of time. We address this issue again in Section 5.2.5.

5.2.4 Retrieving Upper Bound Information

The second primary use of the grid structures is to find an upper bound associated with a particular search node. Recall the situation shown in Figure 5.5, where a particular node has m children. If an upper bound on all leaf children is shown to be poorer than an already-found solution, backtracking can be invoked immediately, precluding any further intersection tests and branching below the current node. The tighter the bound, the more likely that pruning opportunities will be identified and taken advantage of.

However, typically, the better the quality of the bound, the more expensive the cost of each node that is actually expanded. We provide two bounds, one that's cheap and loose, and one that's more expensive but tighter. The idea is to try the cheap one first to catch a large portion of cases, and use the expensive one next in those rarer cases that get by the first pruning attempt. They will be referred to with the following syntax in later algorithm descriptions in Section 5.3:

- `bound = grid[i]->getCheapBound(qbox)`, and

- `bound = grid[i]->getTightBound(qbox)`.

As before, `qbox` is a query box, and `i` is an index indicating a particular objective function. The return value, `bound`, is a simple numerical value.

In the example in Figure 5.9 below, the situation for an arbitrary node at level i is depicted (as in Figure 5.5). The darkened `nodeBox` is the intersection of the `nodePiece` associated with the node, and the parent's `nodeBox`. This intersection region is the query box used for obtaining an upper bound.

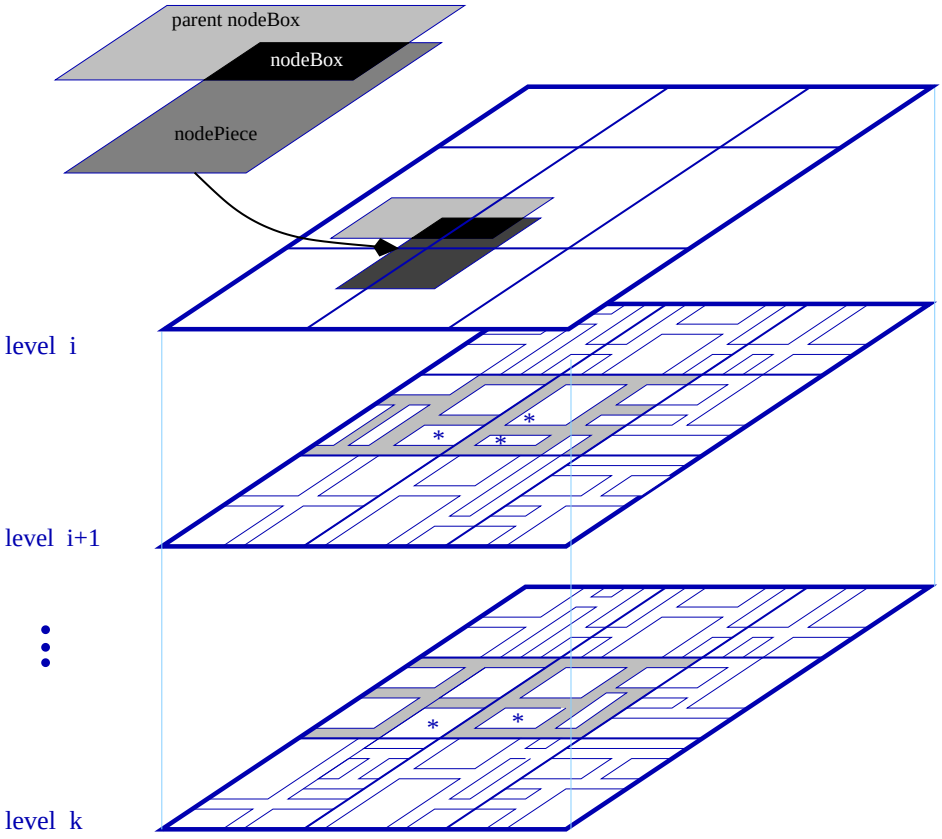


Figure 5.9: Finding an upper bound using the grid structure.

To see how the cheap bound is obtained, let `grid[j]->gridVal(qbox)` be the maximum of the gridVals associated with all grid elements that intersect the given `qbox` for objective function j. In the example in Figure 5.9, there are only two grid elements that intersect the query box at each level. So for a problem with k objective functions, the upper bound for this node would be given by

$$\sum_{j=i+1}^{k} \texttt{grid[j]}\rightarrow\texttt{gridVal(qbox)},$$

where the `qbox` is the `nodebox` associated with the node. This bound is extremely quick to obtain because the values associated with each grid element are set once, when the grid is built, and

populated with pieces from each objective function. It is a looser bound because the `gridVal` for each grid element may be based on a piece that does not in fact intersect the query box.

To obtain a tighter bound, let `grid[j]->pairVal(qbox)` be the maximum weight of the pieces that pair-wise intersect the given `qbox` for objective function j. So for a problem with k objective functions, the tighter upper bound for this node is given by

$$\sum_{j=i+1}^{k} \texttt{grid[j]}\rightarrow\texttt{pairVal(qbox)},$$

where the `qbox` is the `nodeBox` associated with the node in question. Obtaining this bound is typically much slower due to the pairwise intersection tests performed on all pieces found in each grid element. In the example in Figure 5.9, there are 10 pieces at level $i+1$, and 8 pieces at level k that must be pairwise tested for intersection. Note that the returned bound is progressively tighter as the size of the query box shrinks. Since the query box does indeed shrink deeper into the tree, bounding (both kinds) gets tighter as well. If the query box is a point box (containing one point in the decision space), the tight bound is perfect in the piecewise constant case, but not in the piecewise linear case.

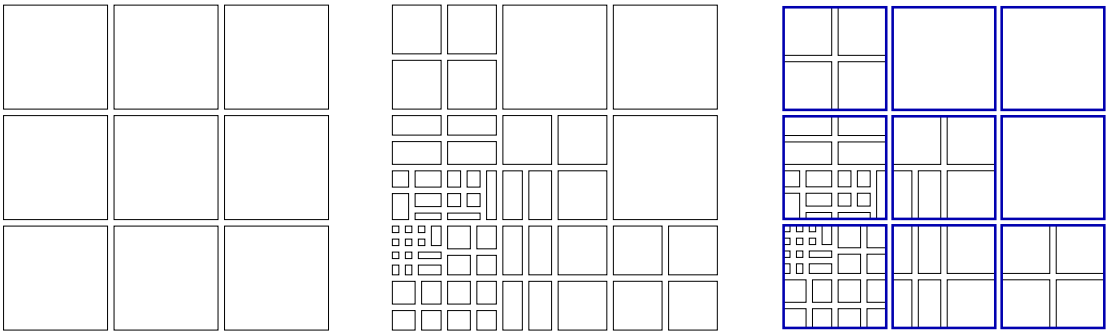
5.2.5 Coordinating a Grid With Its Corresponding Objective Function

Here we tie together the three issues of grid efficiency, grid element layout, and semi-uniform IvP functions. By grid efficiency, we mean the speed and accuracy in retrieving intersection and bounding information from the grid as discussed above.

Semi-uniform IvP functions, first discussed in Section 4.3.3, are created by allocating an initial portion of the total number of pieces to represent the underlying function in a uniform piecewise manner. This idea is depicted in Figures 5.10(a) and 5.10(b). The piece distribution in Figure 5.10(a) is perfectly uniform, using only 9 pieces initially, with the remaining 53 pieces created by splitting the original 9 pieces and their descendents, as shown in Figure 5.10(b). The latter pieces are allocated disproportionately to areas of the function domain producing higher boost in the accuracy representing the underlying function.

Grid efficiency can be dramatically improved if the grid element layout is aligned with the layout of pieces in the IvP function. In creating the IvP function, the piece size used in the initial uniform function is retained and used as the "`gel`" argument in initializing the grid elements as discussed above in Section 5.2.2. The idea is depicted in Figure 5.10(c).

This alignment makes intersection queries quicker because no piece resides in more than one grid element. Recall that the returned list from an intersection query is the concatenation of lists taken from each grid element intersecting the query box. If a piece may reside in more than one grid element, then duplicates must be checked for and removed from the resulting list (see end of Section 5.2.3). The alignment also makes the bounding queries more efficient. When a piece is added to the grid, the upper bound for the grid element is based on the maximum value of the linear interior



(a) Uniform 2D IvP function (b) Semi-uniform IvP function (c) IvP function aligned with grid

Figure 5.10: Aligning the grid with the initial uniformity of an IvP function.

function of the piece. If a portion of the piece lies outside the grid element, then the bound may not be as tight as it could be. This situation is eliminated when the grids are aligned. In general, comparing two grids with the same number of grid elements, each holding the same pieces from the same objective function, the grid with the lower average number of pieces per grid element will be the more efficient grid. Aligning the grid element boundaries with a semi-uniform IvP function typically reduces this ratio dramatically.

5.3 Using the Grid Structure During Search

Having described the grid structure and its uses, we now return to provide better versions of the `IPAL` and `RIPAL` algorithms based on the grid intersection and bounding functions.

5.3.1 Avoiding Intersection Tests by Using the Grid

The version of `IPAL` and `RIPAL` given in Figure 5.11 use the grid associated with each of the k objective functions to avoid as many of the m intersection tests as possible at each node. The grids are created and initialized at the beginning of `IPAL` (lines 1-4) for all but the first objective function. Expanding each node in the first level of the tree (lines 5-7) is done to ensure global optimality, so a grid is not created for the first objective function. As before, `p(i, j)` refers to piece i from objective function j, and there are k objective functions, and m pieces in each function.

```
IPAL()
0. uninitialized = TRUE
1. for(j = 2 to k)
2.    grid[j]→initialize(universe, gel)
3.    for(i = 1 to m)
4.       grid[j]→addPiece(p(i, j))
5. for(i = 1 to m)
6.    nodeBox = p(i, 1)
7.    RIPAL(nodeBox, 2)
8. return(bestBox)
```

(a) IPAL

```
RIPAL(nodeBox, level)
0. if(level == k)
1.   if(uninitialized or nodeBox→wt > best)
2.       best = nodeBox→wt
3.       bestBox = nodeBox
4.       uninitialized = FALSE
5.   return
6. boxset = grid[level]→getBoxes(nodeBox)
7. ibox = boxset→first
8. while(ibox ≠ NULL)
9.   newNodeBox = nodeBox ∩ ibox
10.  RIPAL(newNodeBox, level+1)
11.  ibox = ibox→next
```

(b) RIPAL

Figure 5.11: Versions of `IPAL` and `RIPAL` utilizing intersection information from a grid.

In the version of `RIPAL` in Figure 5.11(b), lines 0-5 remain unchanged from Figure 5.4. Lines 8-11 now reflect that iteration through a linked list is conducted rather than the array of pieces of fixed length m as before. This linked list is the subset of m pieces determined, by a call to the grid in Line 6, to intersect the `nodeBox`. An empty returned `boxset`, or the end of the linked list is detected when `ibox=NULL`. (Note, however, that the `boxset` returned in line 6 can never be empty since the `nodeBox` is guaranteed to be non-empty before expanding the current node, and each point in the decision space is guaranteed to be contained in one piece in each IvP function.)

5.3.2 Pruning by Using Grid Derived Upper Bounds

To utilize the bounding information from the grids, lines 6-15 in Figure 5.12 below are inserted into the previous version of `RIPAL`. The two bounding functions described in Section 5.2.4 are accessed. First the cheap, fast bound is attempted in lines 6-10, followed by the tighter, more time-consuming bound in lines 11-15 if the first one fails to find a pruning opportunity. Note that the loops in lines 6 and 11 go through all remaining levels before the total is checked against the value of `best`. Since a bound from any level may have a negative value, this is necessary. Otherwise the bounding process could be interrupted if the intermediate value became greater than the value of current best solution.

```
RIPAL(nodeBox, level)
0.  if(level == k)
1.    if(uninitialized or nodeBox→wt > best)
2.       best = nodeBox→wt
3.       bestBox = nodeBox
4.       uninitialized = FALSE
5.    return
6.  cheapBound = 0
7.  for(j = level to k)
8.    cheapBound = cheapBound + grid[j]→getCheapBound(nodeBox)
9.  if((nodeBox→wt + cheapBound) ≤ best)
10    return
11  tightBound = 0
12  for(j = level to k)
13    tightBound = tightBound + grid[j]→getTightBound(nodeBox)
14. if((nodeBox→wt + tightBound) ≤ best)
15.   return
16. boxset = grid[level]→getBoxes(nodeBox)
17. ibox = boxset→first
18. while(ibox ≠ NULL)
19.   newNodeBox = nodeBox ∩ ibox
20.   RIPAL(newNodeBox, level+1)
21.   ibox = ibox→next
```

Figure 5.12: New version of RIPAL utilizing the bounding information from a grid.

This version of RIPAL along with the IPAL in Figure 5.11(a) are the versions used in reporting empirical results in Section 5.6. They represent the versions we may refer to later as "IvP solution methods".

5.4 Brute Force Search as an Alternative Solution Method

Brute force search through the decision space is relatively simple, perfectly accurate, but typically too slow for practical purposes. The basic idea is to march blindly from one candidate decision to the next, evaluating each point based on the set of objective functions, and keeping the best decision as the final solution. It is a useful method to implement because it provides a valuable error check of the IvP algorithms (if one is willing to wait long enough, or if the problem is simple enough). Furthermore, the algorithm can be modified to scale back the search to a subset of the decision space as discussed in Section 3.3.2, thus entering the game of accuracy vs. speed tradeoffs.

With available computing power growing so rapidly, and with the subjective nature of accuracy vs. speed tradeoffs, our intention is to show that IvP algorithms completely dominate such brute

force search in all but a fringe set of cases. What is this fringe set? In the case of a full search through the decision space, IvP methods can only guarantee such perfect accuracy if each IvP piece is a *point box*, one for each decision. In this extreme case, the simplicity of search through the decision space, will result in a faster solution. In this section we lay bare the algorithm for brute force search through the decision space, and its variations.

5.4.1 Iterating Through the Decision Space

Here we describe the method of iterating through the decision space. Recall that in Section 3.1.1, the decision space is assumed to be finite and uniformly discrete. Without loss of generality, it can also be assumed that each variable domain contains only integer values and ranges from zero to an integer maximum. The function of interest is `NEXT_PT(ptBox, stepBox)`, which does not return a value but alters the given value of `ptBox`. Both `ptBox` and `stepBox` are of the same syntactic structure of an IvP piece (Section 3.1). The value of `stepBox` determines a subset of the decision space that will

```
NEXT_PT(ptBox, stepBox)
0.  done = FALSE
1.  dim = 1
2.  while(done==FALSE)
3.     val = ptBox→getPT(dim, 0)
4.     newVal = val + stepBox→getPT(dim, 1)
5.     if(newVal ≤ universe→getPT(dim, 1)
6.        ptBox→setPTS(dim, newVal, newVal)
7.        done = TRUE
8.     else
9.        if(dim == n)
10.          ptBox = NULL
11.          done = TRUE
12.       else
13.          for(i = 1 to dim)
14.             ptBox→setPTS(i, 0, 0)
15.          dim = dim + 1
16. return
```

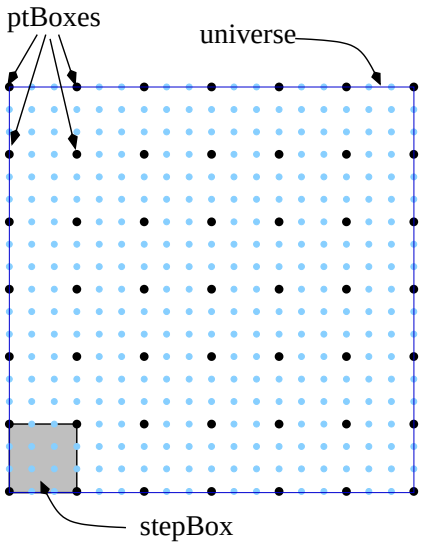


Figure 5.13: The algorithm for iterating through the decision space.

be searched. As indicated in the right side of Figure 5.13, the darkened points are the points in the decision space to be visited. The gap between points is determined by the size of the `stepBox`. If the `stepBox` has a unit length on all sides, then each point in the decision space is visited. In line 5 of `NEXT_PT`, global access to `universe` is assumed. The expressions `piece→setPTS(dim, lowVal, highVal)` and `piece→getPT(dim, high/low)` set and return the high and low interval values for a particular dimension in an IvP piece.

5.4.2 Evaluating a Point in the Decision Space

The evaluation of a given point is indicated by the call to `EVAL_PT(ptBox)` shown below in Figure 5.14. The given point is evaluated by each of the k objective functions indicated by the expression `of(i, ptBox)`. In comparing brute force search to IvP methods, the brute force algorithm utilizes

```
EVAL_PT(ptBox)
0. totalVal = 0
1. for(i = 1 to k)
2.    thisVal = of(i, ptBox)
3.    totalVal = totalVal + thisVal
4. return(returnVal)
```

Figure 5.14: Evaluating a point in the decision space w.r.t. the k objective functions.

the "underlying" objective function if possible. Typically this underlying function can be stated as an analytical expression, whereas a point evaluation in an IvP function requires one to first identify the piece containing the point, and then apply the interior function to the given point. (Recall Figures 3.1 and 3.2.)

5.4.3 The Decision-Space Brute Force Search Algorithm

The brute force algorithm, utilizing the `NEXT_PT` and `EVAL_PT` calls, is given below in Figure 5.15. In

```
BFORCE(stepBox)
0.  for(i = 1 to n)
1.     ptBox→setPTS(i, 0, 0)
2.  best = evalPT(ptBox)
3.  bestPT = ptBox
4.  nextPT(ptBox, stepBox)
5.  while(ptBox ≠ NULL)
6.     currVal = evalPT(ptBox)
7.     if(currVal > best)
8.        best = currVal
9.        bestPT = ptBox
10.    nextPT(ptBox, stepBox)
11. return(bestPT)
```

Figure 5.15: The algorithm for brute force search through the decision space.

lines 0-4, the working best solution is initialized by evaluating the first point in the decision space. The loop in lines 5-10 iterate through the remaining points. When the end of the decision space is reached, the function `NEXT_PT(ptBox, stepBox)` transforms the given `ptBox` into `NULL`.

5.5 Plane Sweep as an Alternative Solution Method

The algorithm by Imai and Asano (1983) for finding the max-clique in an intersection graph is another candidate suitable for solving IvP problems with rectilinear pieces. An intersection graph shown on the right in Figure 5.16 has a vertex corresponding to each rectangle and is connected if the

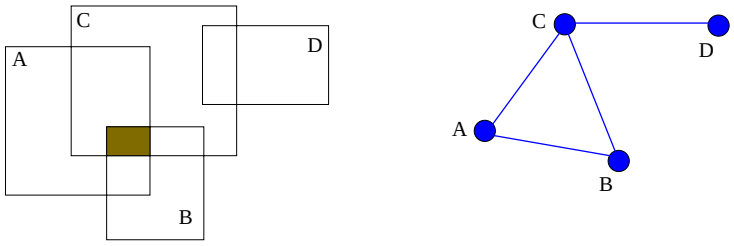
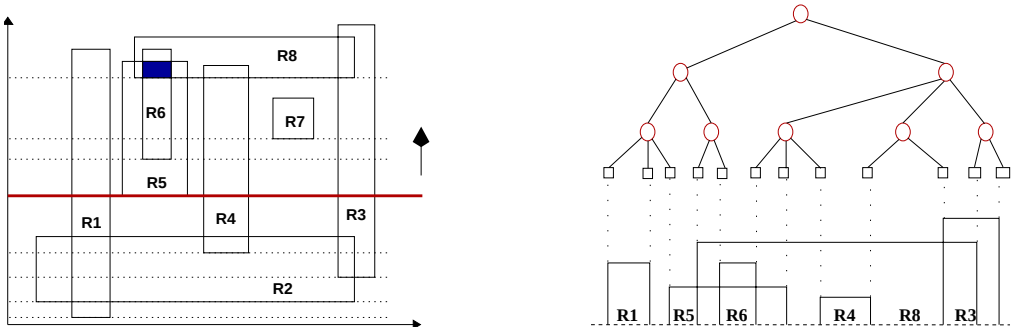


Figure 5.16: A set of rectangles and its corresponding intersection graph.

two rectangles intersect. The problem of finding the max-clique is equivalent to finding the largest set of rectangles that have a non-empty intersection. This can be easily generalized to rectangles with weights to solve rectilinear piecewise constant IvP problems.

5.5.1 Implementation of the Plane Sweep Method

The algorithm by Imai and Asano uses a plane sweep method and a 2-3 tree, depicted below in Figure 5.17, to sweep through all but one dimension stopping at every lower edge of a rectangle and inserting and removing elements from the 2-3 tree that intersect with the sweep line. Since insertions



(a) The plane sweep line (from Imai and Asano (1983))

(b) The 2-3 tree

Figure 5.17: Imai-Asano plane sweep method.

and removals to the tree take $logn$ time, the total time is $O(nlogn)$. In higher dimensions, this generalizes to $O(n^{d-1}logn)$. This algorithm has been fully implemented, for benchmark comparisons to IvP solution methods in Section 5.6.1, and generalized to handle negative-weight rectangles, by adjusting the plane sweep progression to stop at the upper edge of each rectangle in addition to the lower edge.

5.5.2 Limitations of the Plane Sweep Method

The plane sweep algorithm was not designed with IvP problems in mind, but probably in the context of quite different applications, most likely VLSI chip design. In the latter application, there is no a priori knowledge that two given rectangles do not intersect. In IvP problems, we do know that two pieces from the same IvP function do no intersect. The plane sweep method written about in Imai and Asano (1983), and implemented here, does not capitalize on such knowledge. Furthermore, this method does not work properly with IvP pieces with linear interior functions, as well as with pieces with non-rectilinear edges. Despite these limitations, it is a useful method to implement as a correctness check against sufficiently small, rectilinear piecewise constant IvP functions. In Section 5.6.1, benchmark comparisons are made between the two methods.

5.6 Empirical Results

Here we present empirical results regarding the relative effectiveness of the IvP algorithms with respect to the plane sweep method presented above, and with respect to certain changes in problem parameters. The IvP algorithms used in these results are based on the versions of `IPAL` and `RIPAL` shown in Figures 5.11(a) and 5.12 respectively.

As in Chapter 4, the empirical results are based on contrived "ring" functions. See Section 4.2 for details on these functions. The functions found in vehicle control described in the next chapter (Chapter 6) are indeed quite different, but a fair amount of insight into IvP algorithm performance can be gained by using these functions as test data. The parameters of these problems can be twisted in arbitrary, challenging ways that go beyond the problems currently expected to be found in practice. For example, we can observe the solution performance of problems with 25 or 50 objective functions even though we may not currently be able to imagine a vehicle with more than 5 simultaneous behaviors (and thus 5 objective functions). Expanded details for the setup and results of each of the below experiments can be found in Appendix F.

5.6.1 Plane Sweep Search vs. IvP Methods

In this first experiment, the plane sweep algorithm is compared to to IPAL. Since the plane sweep method does not work on pieces with linear interior functions, our test problems are restricted to piecewise constant problems. The results are shown in Figure 5.18.

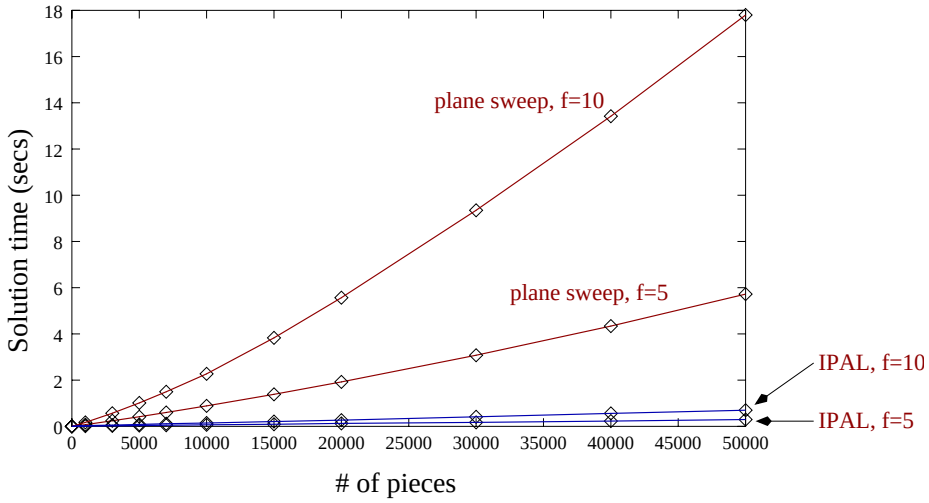


Figure 5.18: The plane sweep algorithm vs. branch and bound.

We created 2D problems with the piece count ranging from 1,000 to 50,000 pieces, 5 and 10 objective functions. The two lines shown for each algorithm represent the set with different amounts of objective functions. The branch and bound results are clearly better. The plane sweep algorithm is known to scale poorly in higher dimensions as discussed in Section 5.5.1.

5.6.2 Solution Time vs. Number of Dimensions

In this experiment seek to see how the solution time of the branch and bound algorithm grows as the number of dimensions grow. For each data point, we created 10 random functions, each with 10,000 pieces, and 5 objective functions, and each based on 5 ring functions. The results are shown in Figure 5.19.

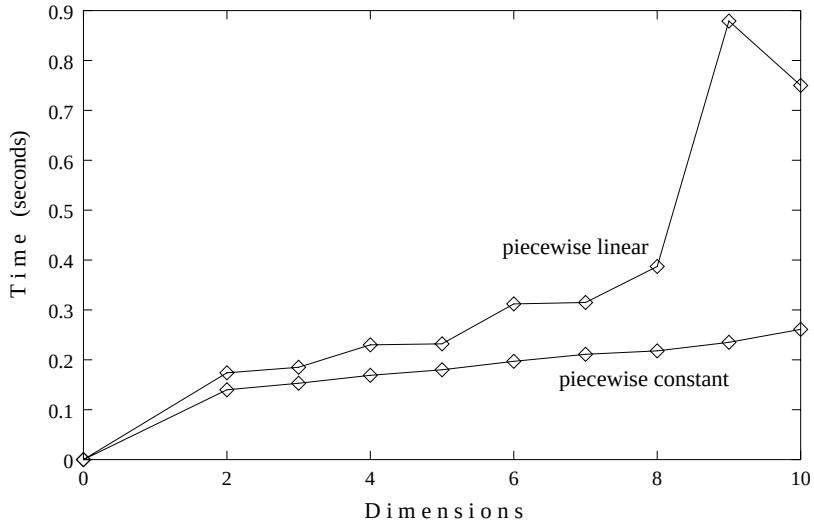


Figure 5.19: IvP solution time vs. number of dimensions.

The anomaly shown above for the 9D piecewise linear case is explained and accounted for in Appendix F.

The two sets of data plotted reflect piecewise constant vs. piecewise linear functions. The piecewise linear functions take more time to solve for at least the following two reasons. First, the representation of pieces with linear interior functions contain more data fields as the number of dimensions grow (a slope for each dimension). The pieces with constant interior functions require a single floating point number regardless of the number of dimensions. Thus there is simply more added overhead involved in dealing with piecewise linear functions as the number of dimensions grow. Secondly, the bounding methods used by the grids in the branch-and-bound algorithm (see Section 5.2.4) are more effective when pieces have constant interior functions.

5.6.3 Solution Time vs. Number of IvP pieces

In this experiment seek to see how the solution time of the branch and bound algorithm grows as the number of pieces grow. For each data point, we created 10 random functions, each in 4 dimensions, and with 5 objective functions. The results are shown in Figure 5.20.

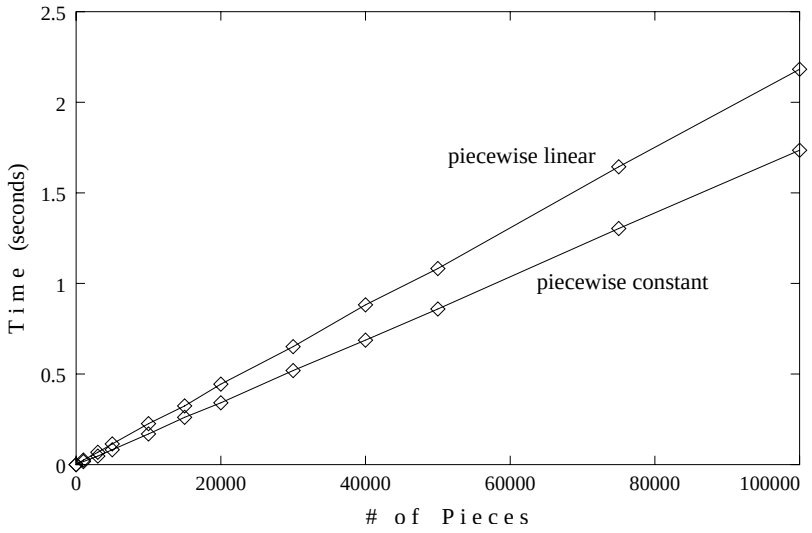


Figure 5.20: IvP solution time vs. number of pieces in each IvP function.

For each sub-experiment of like piece count, the grids were constructed in an appropriate way to keep the pieces per grid-element ratio roughly constant at about 2:1. The details can be found in Section F.4. Its important to note that 80-90% of the total solution time is consumed by initializing the grids before the branch-and-bound process begins. This break-down is also shown in Section F.4, Table F.3. This is significant since, in practice, the grids are initialized by the individual behaviors, which are independent and may be implemented to work in parallel. The dominance of the initialization portion may also largely account for the overall linear growth in time, since each insertion of a piece into a grid takes $O(1)$ time.

5.6.4 Solution Time vs. Number of Objective Functions

In this experiment seek to see how the solution time of the branch and bound algorithm grows as the number of objective functions grow. For each data point, we created 10 random functions, each in 4 dimensions, and with 5,000 pieces. The results are shown in Figure 5.21.

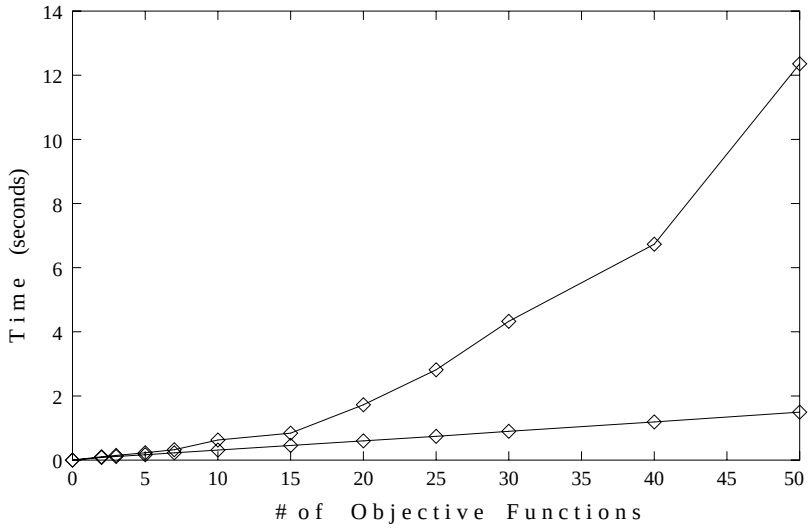


Figure 5.21: IvP solution time vs. number of objective functions.

The nonlinear growth in the piecewise linear case is largely due to the poorer bounds returned by the grids in problems with more objective functions. Recall from Sections 5.2.1 and **??**, that when a piece is inserted to a grid element, the maximum interior point for that piece is added to the running tally for the grid element. There is no effort to keep track of *where* inside that particular piece lies the maximum point.

Chapter 6

IvP and Autonomous Underwater Vehicle Control

In this chapter, we apply the IvP model to problems found in motion planning in autonomous underwater vehicles (AUVs). Previously, in Chapter 2, we identified a need for multi-objective optimization in behavior-based action selection for autonomous vehicles, and argued why existing models do not meet our needs. In Chapters 3 through 5 we defined a new model, IvP, and demonstrated its expressive power and solution power on a variety of contrived "randomly generated" problems. At this point, we wish to bring these two topics back together and show that IvP does indeed fill the need we identified earlier, and that the IvP model and algorithms are tractable in the types of problems we expect to actually find in practice.

The road-map for this is as follows. We first focus on a particular, challenging scenario in vehicle motion planning, described below in Section 6.1. From this, a set of core behaviors are identified that will, collectively, enable the vehicle to perform its mission in the scenario. We then show that the IvP functions corresponding to each of the core behaviors can be accurately and quickly built. And finally, in Chapter 7, we show, in vehicle simulation, that the resulting IvP problems can be solved quick enough to satisfy control loop requirements, and that the vehicle controller composed of these behaviors meets the behavioral demands of the scenario.

6.1 The Vehicle Control Scenario

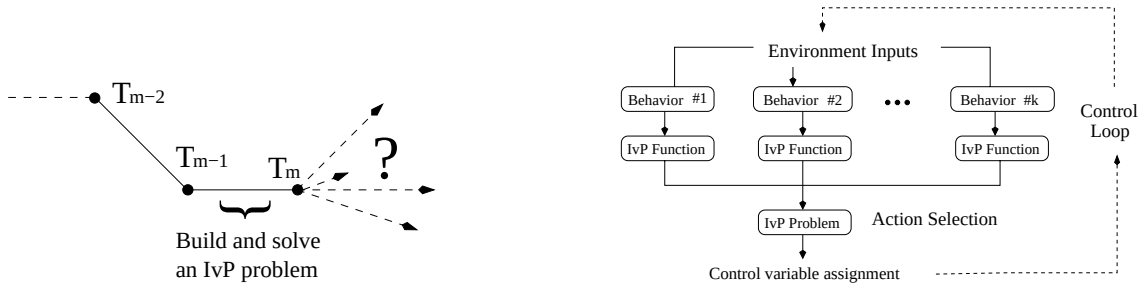
The mission assigned to an underwater vehicle strongly shapes the navigation complexity and criteria for success. While many problems are similar between commercial and military AUVs, there is a stronger emphasis in military vehicles in reasoning about other nearby moving vessels (Fletcher, 2000; Wernli, 2001). Military AUVs (more commonly referred to as unmanned underwater vehicles (UUVs)) are typically designed to operate in congested coastal situations, where a near-collision or

mere detection by another vessel can jeopardize the AUV.

The scenario considered in this chapter therefore centers around the need to consider preferred relative positions to a moving contact, while simultaneously transiting to a destination as quickly and directly as possible. By "preferred relative position", we primarily mean collision avoidance, but use this term also in reference to other objectives related to relative position. These include the refinement of a solution on a detected contact, the avoidance of detection by another contact, and the achievement of an optimal tactical position should an engagement begin with the contact.

6.1.1 IvP and the Vehicle Control Loop

The situation considered here involves a vehicle moving through time and (ocean) space, where periodically, at fixed time intervals, a decision is made as to how to next control the vehicle. As shown in Figure 6.1(a), the "next" decision occurs at time T_m, while the decision making process is conducted in the time interval $[T_{m-1}, T_m]$. Each iteration of this control loop involves the building and solving of an IvP problem as shown in Figure 6.1(b). Each IvP function is derived from an



(a) The vehicle moving through time and space

(b) The control loop structure

Figure 6.1: An IvP problem is created and solved on each iteration of the control loop.

individual behavior. Each behavior has access to the information in the environment that it finds relevant in building its IvP function. Each IvP function is defined over a common decision space, where each decision precisely spells out the next action for the vehicle to implement starting at time T_m. In what follows, we describe this environment in which the vehicle operates and from which the behaviors feed. We also describe the decision space (alternatively referred to as the action space) and the rationale for using the decision variables chosen here.

6.1.2 The Vehicle Environment

The information that composes the vehicle's relevant environment can be divided into the following four groups: a) bathymetry data, b) destination information, c) ownship position information, and d) contact position information. The bathymetry data represents an assumed map of the environment, telling us what is reachable from where, and at which depths. To varying degrees of resolution, this

data is publicly available for the vast majority of the ocean world. In Figure 6.2 on the left, the bathymetry data is shown for a particular region near Florida and the Bahamas, with deeper areas in the ocean represented by darker shades of grey. The destination is simply given by a particular latitude-longitude value.

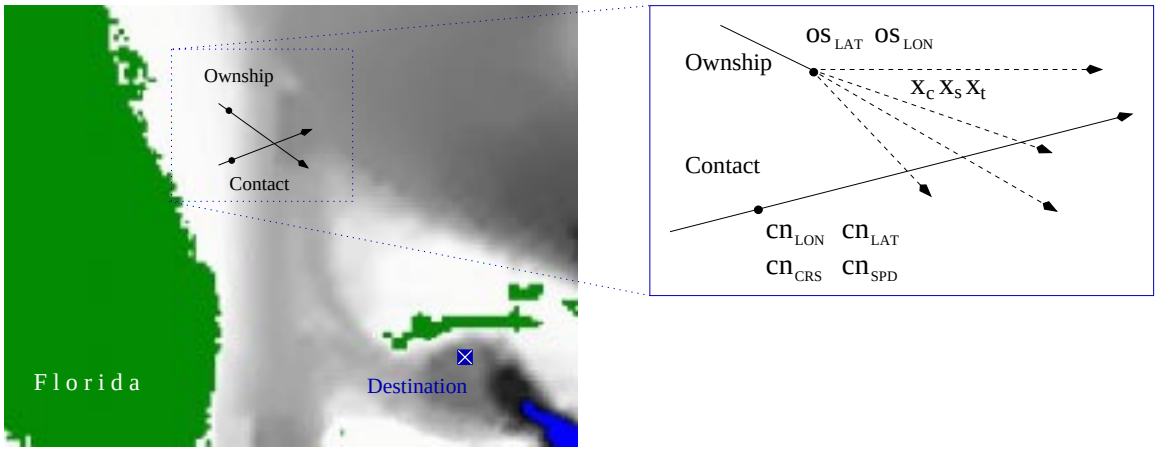


Figure 6.2: The Scenario with ownship and moving contact.

The position information for ownship is given by the terms $\texttt{os}_{\texttt{LAT}}$ and $\texttt{os}_{\texttt{LON}}$, as indicated on the right in Figure 6.2. This is the expected vehicle position at time T_m, based on its position at time T_{m-1} and the choice of course and speed executed at T_{m-1}. Likewise, the position for a contact is given by $\texttt{cn}_{\texttt{LAT}}$ and $\texttt{cn}_{\texttt{LON}}$, based on the contact's observed course and speed at time T_{m-1}. In addition, the terms $\texttt{cn}_{\texttt{CRS}}$ and $\texttt{cn}_{\texttt{SPD}}$ indicate the expected course and speed of the contact at time T_m, which is simply the previous course and speed. These six variables are summarized in Table 6.1 below.

$\texttt{os}_{\texttt{LAT}}$	Latitude position of ownship at T_m
$\texttt{os}_{\texttt{LON}}$	Longitude position of ownship at T_m
$\texttt{os}_{\texttt{CRS}}$	Course of ownship at T_m
$\texttt{os}_{\texttt{SPD}}$	Speed of ownship at T_m
$\texttt{cn}_{\texttt{LAT}}$	Latitude position of contact at T_m
$\texttt{cn}_{\texttt{LAT}}$	Longitude position of contact at T_m
$\texttt{cn}_{\texttt{CRS}}$	Course of contact at T_m
$\texttt{cn}_{\texttt{SPD}}$	Speed of contact at T_m

Table 6.1: Given ownship position and contact solution.

During the time interval $[T_{m-1}, T_m]$, the contact is assumed to be on a straight linear track. The calculated ownship maneuver would still be carried out regardless of a change in course or speed made by the contact in this time interval. Should such a change occur, the new $\texttt{cn}_{\texttt{CRS}}$ and $\texttt{cn}_{\texttt{SPD}}$ would be noted, the next $\texttt{cn}_{\texttt{LAT}}$ and $\texttt{cn}_{\texttt{LON}}$ calculated, and the process of determining the maneuver at time T_{m+1} begun. The implementation of a tight control loop, and the willingness to repeatedly reconsider the next course of action, ensures that the vehicle is able to quickly react to changes in

its perceived environment.

6.1.3 The Vehicle Control Variables and Decision Space

In the scenario considered here, the following three decision variables are used to control the vehicle: x_c = `course`, x_s = `speed`, and x_t = `time`. They are summarized, with their corresponding domains and resolutions in Table 6.2 below.

Name	Meaning	Domain	Resolution
x_c	Ownship course starting at time T_m	$[0, 359]$	1 degree
x_s	Ownship speed starting at time T_m	$[0, 30]$	1 knot
x_t	Intended duration of the next ownship leg	$[1, 90]$	1 minute

Table 6.2: The three decision variables, their domains, and resolution.

The selection of these three decision variables, and the omission of others, reflects a need to present both a sufficiently simple scenario here, as well as a sufficiently challenging motion planning problem. The omission of variables for controlling vehicle depth, for example, may seem strange since we are focusing on AUV's. However, the five objective functions presented in Sections 6.2 through 6.6 focus on using the IvP model to solve the particularly challenging problem of shortest/quickest path navigation in the presence of moving obstacles.

Although reasoning about vehicle depth is critically important for successful AUV operation, none of the objective functions we implement here involve depth. In the scenario described, it is assumed that the depth remains fixed at a preset level. The same holds true for other important control variables, namely the ones that control the rate of change in course, speed or depth. Again for the sake of simplicity, it is assumed that a course or speed change given by $\langle x_c, x_s \rangle$ will take place at some reasonable rate. Alternatively, we can regard such maneuvers as happening instantaneously, and include the error that results from this erroneous assumption into general unpredictability of executing an action in a world with limited actuator precision. Certainly, the decision space will grow in size and complexity as more realistic scenarios are considered. Recall that this is one of the motivations for the IvP model.

Even when limited to the three variables above, with their domains and resolutions, the decision space contains $360 \times 31 \times 90 = 1,004,400$ elements. By comparison, none of the decision spaces considered by Pirjanian (1998), Rosenblatt (1997), or Riekki (1999) contained more than 1,000 elements, even if those decision spaces were composed as the Cartesian product of their variable domains, which they were not, except by Riekki (1999).

6.2 Behavior 1: Safest Path

The objective of the *safest path* behavior is to prevent ownship from coming dangerously close to a particular contact, and is defined over the three decision variables x_c, x_s, and x_t. We describe how to build an IvP function, $f_{\text{IvP}}(x_c, x_s, x_t)$, based on an underlying function, $f_{\text{CPA}}(x_c, x_s, x_t)$. The latter function is based on the closest point of approach, (CPA), between the two vehicles during a maneuver, $\langle x_c, x_s, x_t \rangle$, made by ownship. This function is calculated in a three step process:

[1] Determine the point in time when the closest point of approach occurs, x'_t.
[2] Calculate the distance between vehicles at this time x'_t.
[3] Apply a utility metric to this distance.

After discussing how $f_{\text{CPA}}(x_c, x_s, x_t)$ is calculated, the creation of $f_{\text{IvP}}(x_c, x_s, x_t)$ from this function is discussed in Section 6.2.3.

6.2.1 Determining When the Closest Point of Approach Occurs

To calculate $f_{\text{CPA}}(x_c, x_s, x_t)$, we first need to find the point in time, x'_t, in the interval $[0, x_t]$, when the CPA occurs. To do this, we need expressions telling us where ownship and the contact are at any point in time, as well as an expression for their relative distance. Recall that at time, T_m, ownship will be at a certain relative position to the contact, and after a particular maneuver, given by $\langle x_c, x_s, x_t \rangle$, will be at a new point in the ocean and at a new relative position. For ownship, the new latitude and longitude position is given by:

$$f_{\text{LAT}}(x_c, x_s, x_t) = \cos(x_c)(x_s)(x_t) + \text{os}_{\text{LAT}}$$
$$f_{\text{LON}}(x_c, x_s, x_t) = \sin(x_c)(x_s)(x_t) + \text{os}_{\text{LON}}.$$

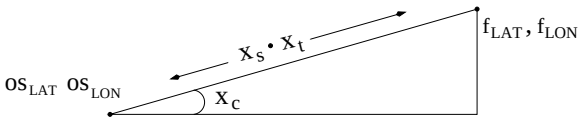


Figure 6.3: Position after $\langle x_c, x_s, x_t \rangle$.

The resulting new contact position is similarly given by the following two functions:

$$g_{\text{LAT}}(x_t) = \cos(\text{cn}_{\text{CRS}})(\text{cn}_{\text{SPD}})(x_t) + \text{cn}_{\text{LAT}}$$
$$g_{\text{LON}}(x_t) = \sin(\text{cn}_{\text{CRS}})(\text{cn}_{\text{SPD}})(x_t) + \text{cn}_{\text{LON}}.$$

The latter two functions are defined only over x_t since the contact's course and speed are assumed not to change from their values of cn_{CRS} and cn_{SPD}. Note these four functions ignore earth curvature. The distance between ownship and the contact, after a maneuver $\langle x_c, x_s, x_t \rangle$ is expressed as:

$$dist^2(x_c, x_s, x_t) = (f_{\text{LAT}}(x_c, x_s, x_t) - g_{\text{LAT}}(x_t))^2 + (f_{\text{LON}}(x_c, x_s, x_t) - g_{\text{LON}}(x_t))^2.$$

Barring the situation where the two vehicles are at identical course and speed, the CPA is at a unique minimum point in the above function. We find this stationary point by expanding this function, collecting like terms, and taking the first derivative with respect to x_t, setting it to zero, and solving for x_t. By expanding and collecting like terms we get:

$$dist^2(x_c, x_s, x_t) = k_2 x_t^2 + k_1 x_t + k_0,$$

where

$$\begin{aligned}
k_2 = \;& \cos^2(x_c) \cdot x_s^2 - 2\cos(x_c) \cdot x_s \cdot \cos(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}} + \cos^2(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}}^2 + \\
& \sin^2(x_c) \cdot x_s^2 - 2\sin(x_c) \cdot x_s \cdot \sin(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}} + \sin^2(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}}^2 \\
k_1 = \;& 2\cos(x_c) \cdot x_s \cdot \mathtt{os_{LAT}} - 2\cos(x_c) \cdot x_s \cdot \mathtt{cn_{LAT}} - 2\mathtt{os_{LAT}} \cdot \cos(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}} + \\
& 2\cos(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}} \cdot \mathtt{cn_{LAT}} + 2\sin(x_c) \cdot x_s \cdot \mathtt{os_{LON}} - 2\sin(x_c) \cdot x_s \cdot \mathtt{cn_{LON}} - \\
& 2\mathtt{os_{LON}} \cdot \sin(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}} + 2\sin(\mathtt{cn_{CRS}}) \cdot \mathtt{cn_{SPD}} \cdot \mathtt{cn_{LON}} \\
k_0 = \;& \mathtt{os_{LAT}}^2 - 2\mathtt{os_{LAT}} \cdot \mathtt{cn_{LAT}} + \mathtt{cn_{LAT}}^2 + \mathtt{os_{LON}}^2 - 2\mathtt{os_{LON}} \cdot \mathtt{cn_{LON}} + \mathtt{cn_{LON}}^2
\end{aligned}$$

From this we have:

$$dist^2(x_c, x_s, x_t)' = 2k_2 x_t + k_1.$$

We note that the distance between two objects cannot be negative, so the point in time, x_t', when $dist^2(x_c, x_s, x_t)$ is at its minimum is the same point where $dist(x_c, x_s, x_t)$ is at its minimum. Also, since there is no "maximum" distance between two objects, a point in time, x_t', where $2k_2 x_t + k_1 = 0$ must represent a minimum point in the function $dist(x_c, x_s, x_t)$. Therefore x_t' is given by:

$$x_t' = \frac{-k_1}{2k_2}.$$

If $x_t' < 0$, meaning the closest point of approach occurred prior to the present, we set $x_t = 0$, and if $x_t' > x_t$, we set $x_t' = x_t$. When ownship and the contact have the same course and speed, i.e., $x_c = \mathtt{cn_{CRS}}$ and $x_s = \mathtt{cn_{SPD}}$, then k_1 and k_2 equal zero, and x_t' is set to zero, since their relative distance will not change during the time interval $[0, x_t]$.

6.2.2 Calculating the CPA and Applying a Metric

Having identified the time, x_t', at which the closest point of approach occurs, calculating this corresponding distance is a matter of applying the distance function, given above, to x_t'.

$$\mathtt{cpa}(x_c, x_s, x_t) = dist(x_c, x_s, x_t').$$

The actual objective function reflecting the safest-path behavior, $f_{\text{CPA}}(x_c, x_s, x_t)$, depends on both the CPA value and a utility metric relating how good or bad particular CPA values are with respect to goals of the safest-path behavior. Thus $f_{\text{CPA}}(x_c, x_s, x_t)$ will have the form:

$$f_{\text{CPA}}(x_c, x_s, x_t) = \texttt{metric}(\texttt{cpa}(\texttt{x}_\texttt{c}, \texttt{x}_\texttt{s}, \texttt{x}_\texttt{t})).$$

We first consider the case where $f_{\text{CPA}}(x_c, x_s, x_t)$ represents a "collision-avoidance" objective function. In a world with perfect knowledge and perfectly executed actions, a constraint-based approach to collision avoidance would be appropriate, resulting in $\texttt{metric}_\texttt{a}(d)$ below, where d is the CPA distance, and $-M$ is a sufficiently large negative number acting as $-\infty$. Allowing for error, one could instead

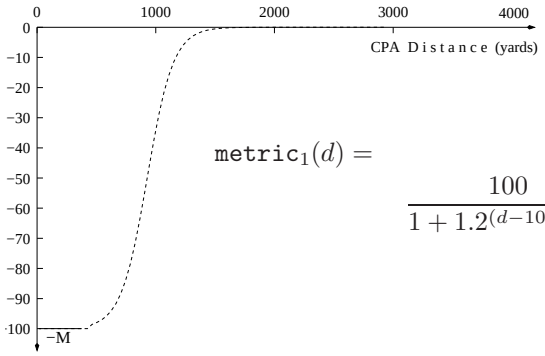
- $\texttt{metric}_\texttt{a}(d) =$ -M if $d = 0$; 0 otherwise

or,

- $\texttt{metric}_\texttt{b}(d) =$ -M if $d \leq 300$; 0 otherwise

use $\texttt{metric}_\texttt{b}(d)$ where maneuvers that result in CPA distances of less than 300 yards are treated as "collisions" to allow room for error, or a buffer zone.

Instead, we use a metric that recognizes that this collision safety zone is grey, or fuzzy. Under certain conditions, distances that would otherwise be avoided, may be allowed if the payoff in other goals is high enough. Of course, some distances remain intolerable under any circumstance. This metric is shown below in Figure 6.4.



$$\texttt{metric}_1(d) = \begin{cases} -M & \text{if } d \leq 150 \\ \dfrac{100}{1 + 1.2^{(d-1000)/20}} & \text{otherwise} \end{cases}$$

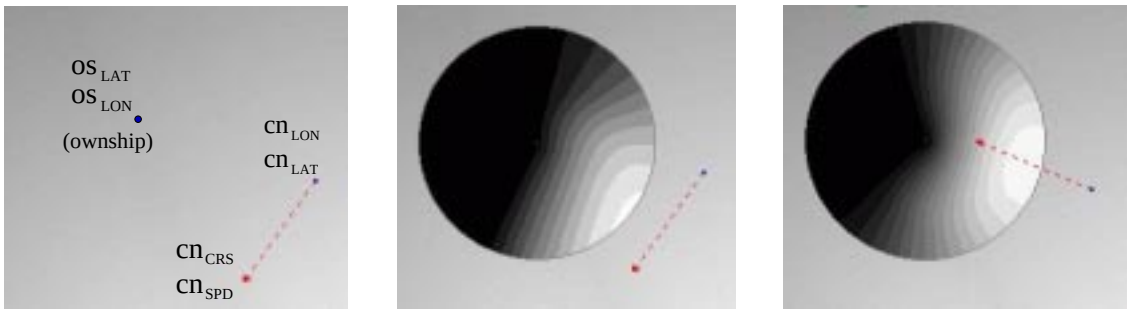
Figure 6.4: A collision-avoidance metric based on closest-point-of-approach distance.

A sigmoid-shaped function is used here because it acts as a "soft" step function.

Having specified a function to compute the CPA distance and a utility metric based on the CPA distance, the specification of $f_{\text{CPA}}(x_c, x_s, x_t)$ is complete. Based on this function, we then build the function $f_{\text{IvP}}(x_c, x_s, x_t)$.

6.2.3 Building the Corresponding IvP Function

Now that $f_{\text{CPA}}(x_c, x_s, x_t)$ has been defined, we wish to build a version of $f_{\text{IvP}}(x_c, x_s, x_t)$ that closely approximates this function. As mentioned in Chapter 4, the idea is to create as accurate a representation as possible, as quickly as possible, using as few pieces as possible. This in itself is a non-trivial multi-objective problem. Fortunately, fairly naive approaches to building this function appear to work well in practice, with additional room for doing much better given more thought and design effort. To begin with, we create a piecewise uniform version of $f_{\text{IvP}}(x_c, x_s, x_t)$ using 1000 pieces. This function looks similar to the ones shown in Figure 6.5. The situation depicted in Figure 6.5(a) shows the given positions of ownship and a contact, as well as the contact's motion vector (215° and



(a) (b) (c)

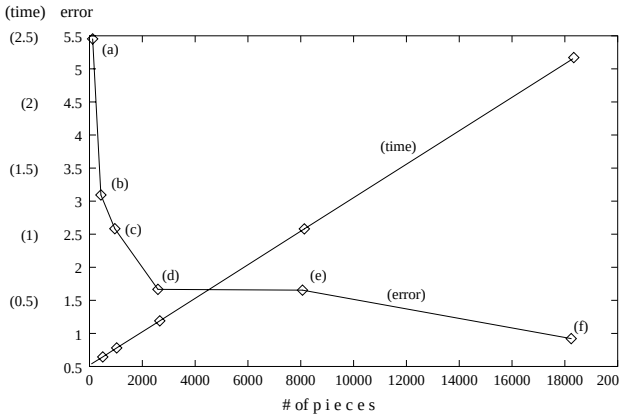
Figure 6.5: (a) A particular situation (b) The resulting $f_{\text{IvP}}(x_c, x_s, x_t)$ objective function, and (c) A different $f_{\text{IvP}}(x_c, x_s, x_t)$ for a different contact course.

20 knots). Since $f_{\text{IvP}}(x_c, x_s, x_t)$ is a 3D function, the view in Figures 6.5(b) and 6.5(c) are a slice of the 3D function for a fixed speed, in this case, the maximum ownship speed of 30 knots. The darker points indicate values closer to zero, and the lighter points are increasingly negative, indicating near collision courses.

Building $f_{\text{IvP}}(x_c, x_s, x_t)$ with 1000 pieces as shown above takes roughly 0.12 seconds, but the questions of acceptable accuracy, time, and piece-count are difficult to respond to with precise answers. The latter two issues of creation time and piece-count are tied to the tightness of the vehicle control loop. This makes it possible to work backward from the control loop requirements to bound the creation time and piece-count. However, the control loop time is also application dependent.

The most difficult issue is knowing when the function $f_{\text{IvP}}(x_c, x_s, x_t)$ is an acceptably accurate representation of $f_{\text{CPA}}(x_c, x_s, x_t)$. Recall however, that there is a large degree of subjectivity in the metric component of $f_{\text{CPA}}(x_c, x_s, x_t)$ (Figure 6.4). Although it is difficult to pinpoint, at some point the error introduced in approximating $f_{\text{CPA}}(x_c, x_s, x_t)$ with $f_{\text{IvP}}(x_c, x_s, x_t)$ becomes overshadowed by the subjectivity involved in $f_{\text{CPA}}(x_c, x_s, x_t)$. Another tool, used in practice here, is to experiment with different versions of $f_{\text{IvP}}(x_c, x_s, x_t)$ and noting when the poorer versions begin to adversely affect vehicle behavior.

The trends depicted in Figure 6.6 below can provide some valuable insight as to where the best accuracy-speed tradeoffs may be found. In this figure, six different versions of $f_{\text{IvP}}(x_c, x_s, x_t)$ are compared with respect to the number of pieces in each function, the creation time, and the error associated with each.



	piece size			No. pieces			total pieces
	x_c	x_s	x_t	x_c	x_s	x_t	
(a)	30	16	20	12	2	5	120
(b)	20	11	12	18	3	8	432
(c)	15	8	10	24	4	10	960
(d)	10	6	8	36	6	12	2592
(e)	5	5	6	72	7	16	8064
(f)	3	4	5	120	8	19	18240

Figure 6.6: Creation time and accuracy vs. piece count for six different uniform piecewise functions.

The six versions are all uniform piecewise functions built with the parameters shown in the table on the right. In the first version of $f_{\text{IvP}}(x_c, x_s, x_t)$, (a), each piece has the dimension $x_c = 30$, $x_s = 16$, and $x_t = 20$. The number of pieces in the x_c dimension will therefore be $\lceil 360/30 \rceil = 12$, and likewise for x_s and x_t. The resulting total number of pieces is plotted on the x-axis in the figure.

The creation of each of these six IvP functions proceeds by simply dividing the decision space into the respective uniform pieces, and calculating the appropriate interior linear function. The time of creation is therefore linear with respect to the piece count as shown. The reported error value is calculated by averaging the absolute error between $f_{\text{CPA}}(x_c, x_s, x_t)$ and $f_{\text{IvP}}(x_c, x_s, x_t)$ for 1000 randomly chosen sample points. We notice a distinct elbow in the error curve roughly at (d). This information, combined with aspiration levels on the time, error and piece count, will lead to an appropriate decision as to precisely how $f_{\text{IvP}}(x_c, x_s, x_t)$ will be constructed. Although this kind of analysis cannot be done on each iteration of the control loop, enough insight can be obtained to choose a piece-count that works sufficiently well in all situations.

6.3 Behavior 2: Shortest Path

The *shortest path* behavior is concerned with finding a path of minimal distance from the current position of the vehicle $\langle \texttt{os}_{\texttt{LAT}}, \texttt{os}_{\texttt{LON}} \rangle$ to a particular destination $\langle \texttt{d}_{\texttt{LAT}}, \texttt{d}_{\texttt{LON}} \rangle$. As with the previous behavior, the aim is to produce an IvP function $f_{\text{IvP}}(x_c, x_s, x_t)$ that not only indicates which next maneuver(s) are optimal with respect to the behavior's goals, but evaluates *all* possible maneuvers in this regard. The primary difference between this behavior and the previous behavior, is that here, $f_{\text{IvP}}(x_c, x_s, x_t)$ is piecewise defined over the latitude-longitude space rather than over the decision space.

The function $f_{\text{IvP}}(x_c, x_s, x_t)$, as in other behaviors, is created during each iteration of the control loop, and must be created quickly. In the shortest path behavior, an intermediate function, $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$, is created once, off-line, for a particular destination, and gives the shortest-path distance to the destination given a point in the ocean, $\langle \texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}} \rangle$. The creation of $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ is described below in Section 6.3.2. This function in turn is built upon a third function, $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$, which returns a depth value for a given point in the ocean, and is described below in Section 6.3.1.

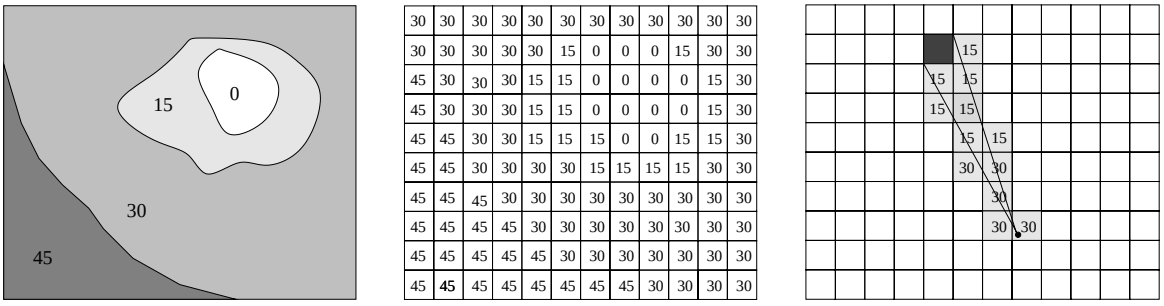
6.3.1 Creating a Piecewise Defined Bathymetry Function

The function $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ is a piecewise constant function over the latitude-longitude space, where the value inside each piece represents the *shallowest* depth within that region. This function is formed in a manner similar to that of building IvP functions, described in Section 4.1. The "underlying" function in this case is a large file of bathymetry data, where each line is a triple: $\langle \texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}}, \texttt{depth} \rangle$. These bathymetry files can be obtained for any particular region of the ocean from the Naval Oceanographic Office Data Warehouse, with varying degrees of precision, i.e., density of data points. The resulting function $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ looks like Figure 6.7 when rendered, with whiter colors representing more shallow waters.

Figure 6.7: A bathymetry function for a region near Florida and Grand Bahama Island.

The primary purpose of the $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ function is to provide a quick and convenient means

for determining if one point in the ocean is directly reachable from another. Consider the example function, $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$, shown in Figure 6.8(b), which is an approximation of the bathymetry data rendered in Figure 6.8(a). From this data, we can answer the query depicted in Figure 6.8(c), which

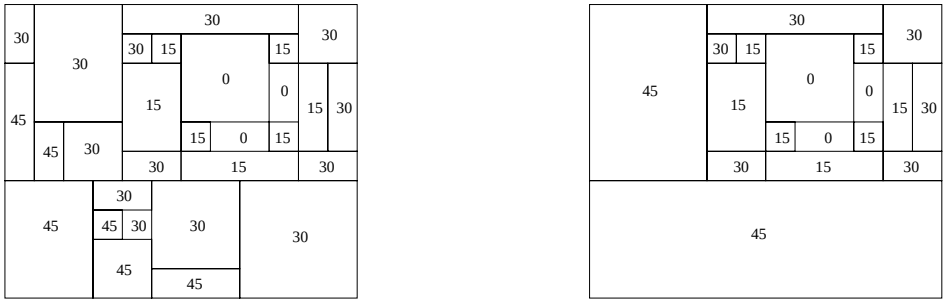


(a) Bathymetry map (b) $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ (c) Example query

Figure 6.8: Determining reachability, for a given depth, using bathymetry data.

returns "yes" if the darkened point is reachable from all points inside the darkened square, for a given depth. In this example, the answer is "yes" for depths in the range $[0, 14]$, and "no" otherwise.

The function $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ is built by using the function $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ and performing many of the above such queries. The accuracy in representing the underlying bathymetry data is enhanced by using finer lat-lon pieces. However, the query time is also increased with more pieces, since all pieces between the two points must be retrieved and tested against the query depth. (Actually, just finding one that triggers a "no" is sufficient, but to answer "yes", all must be tested.) The example function $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ shown above in Figure 6.8(b) uses a uniform piecewise function. An equivalent non-uniform function is shown in Figure 6.9(a), with no loss in precision, by combining neighboring pieces with similar values. Further consolidation can be done if a range of operating



(a) Consolidating adjacent like pieces (b) Consolidating based on a depth

Figure 6.9: Non-uniform representations of $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$.

depth for the vehicle is known a priori. For example, if the vehicle will travel no deeper than 30 meters, then the function in Figure 6.9(a) can be replaced with the one in Figure 6.9(b) with no

loss in precision, since pieces with depths of 30 and 45 meters are functionally equivalent when the vehicle is resticted to depths less than 30 meters.

6.3.2 Creating a Piecewise All-Sources Shortest Path Function

The function $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ is a piecewise linear function over the latitude-longitude space, where the value inside each piece represents the shortest path distance to the destination $\langle \texttt{d}_{\texttt{LAT}}, \texttt{d}_{\texttt{LON}} \rangle$, given a bathymetry function, $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$, and a specific operating depth. We consider here only simple linear distance, but recognize that consideration of other factors, such as preferred depth, current flow, and proximity to obstacles with uncertainty will provide a more robust implementation. These factors are discussed for example by Reif and Sun (2000, 2001).

The algorithm used in our simulation to calculate $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$, given a destination, operating depth, and $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ function, is provided in Appendix A. The resulting function, $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$, looks like Figure 6.10 when rendered, for the same region shown in Figure 6.7, and for a particular destination and depth (in this case zero). The lighter points indicate greater

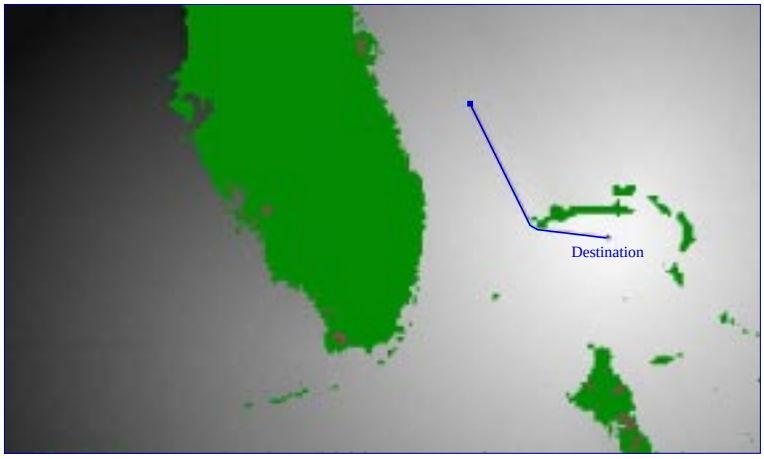


Figure 6.10: $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ for a particular region, depth, and destination.

proximity to the destination. Since the shortest distance for each point is based on a particular set of waypoints composing the shortest path, we also store the next waypoint with each point in latitude-longitude space. This forms a linked list from which a full set of waypoints can be reconstructed for any given start position. An example set of waypoints is shown above in Figure 6.10, for a particular starting point and destination.

6.3.3 Creating a Vehicle-centric IvP Shortest Path IvP Function

Once the function $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ has been created for a particular destination and depth, the function $f_{\text{IvP}}(x_c, x_s, x_t)$ for a given ownship position can be quickly created. Like $\texttt{bathy}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ and $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$, this function is defined over the latitude-longitude space, but the function

$f_{\text{IvP}}(x_c, x_s, x_t)$ is defined only over the points reachable within one maneuver. A distance radius is determined by the maximum values for x_s and x_t. The objective function, $f_{\text{IvP}}(x_c, x_s, x_t)$, produced by this behavior ranks way-points based on the additional distance, over the shortest-path distance, that would be incurred by travelling through them. An example is shown below in Figure 6.11. At the center of the circle is ownship, and darker points indicate way-points with little or no "detour distance".

Figure 6.11: $f_{\text{IvP}}(x_c, x_s, x_t)$ for a particular ownship position and underlying $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$.

For each piece in $f_{\text{IvP}}(x_c, x_s, x_t)$, the linear interior function represents a detour distance calculated using three components. The first two are linear functions in the piece representing the distance to the destination, and the distance to the current ownship position, as in Figure 6.12 below. The

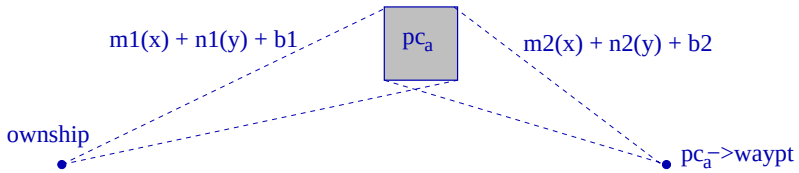
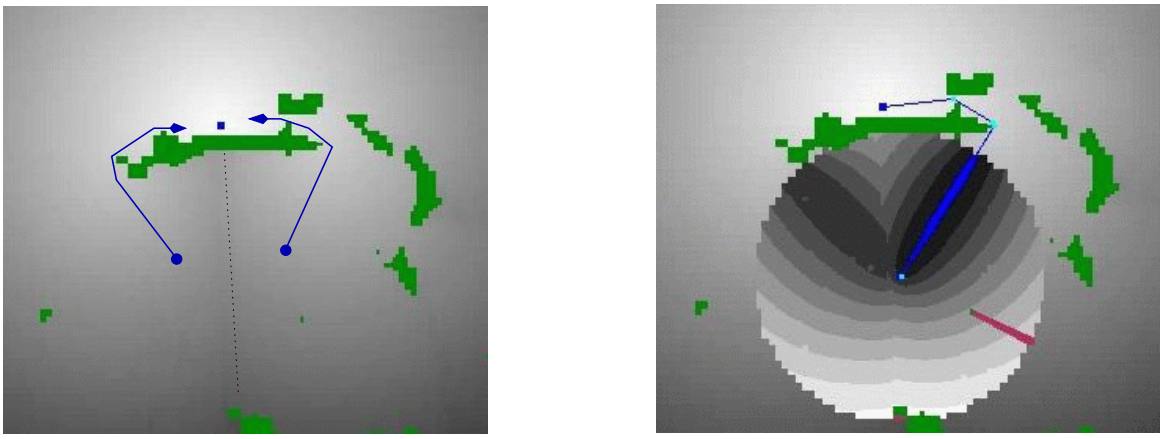


Figure 6.12: Two key linear functions in determining detour distance.

third component is simply the distance from the current ownship position to the destination, given by $\texttt{spath}(\texttt{os}_{\texttt{LAT}}, \texttt{os}_{\texttt{LON}})$. Thus, the linear function representing the detour distances for all points $\langle x, y \rangle$ in a given piece, is given by: $(m_1 + m_2)(x) + (n_1 + n_2)(y) + b_1 + b_2 - \texttt{spath}(\texttt{os}_{\texttt{LAT}}, \texttt{os}_{\texttt{LON}})$. A utility metric is then applied to this result to both normalize the function $f_{\text{IvP}}(x_c, x_s, x_t)$, and allow a nonlinear utility to be applied against a range of detour distances.

The objective functions built by the shortest path behavior may reflect alternative paths that closely missed being the shortest, from a given position. In Figure 6.13(a) below, the shortest path from positions just south of the Grand Bahama Island to the destination just north of the island will proceed either east or west depending on the starting position. A north-south line at roughly $78°30'W$ determines the direction of the shortest path. When ownship is nearly on this line, as



(a) Shortest path depends on start position

(b) Objective function reflects alternatives.

Figure 6.13: Shortest path and alternative near-shortest paths.

in Figure 6.13(b), the resulting objective function, $f_{\text{IvP}}(x_c, x_s, x_t)$, reflects both alternative paths. For the situation depicted in Figure 6.13(b), the shortest path proceeds east around the island. Although positions north-west of ownship's current position represent a significant detour from the true shortest path, they are still ranked highly due to the alternative, near-shortest path. The presence of alternatives is important when the behavior needs to cooperate with another behavior that may have a good reason for *not* proceeding east.

The three functions in this behavior are coordinated to allow repeated construction of $f_{\text{IvP}}(x_c, x_s, x_t)$ very quickly, since it needs to be built and discarded on each iteration of the control loop. The relationships are summarized below in Table 6.3.

stable	$\texttt{bathy}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$	slow
$\updownarrow$	$\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$	$\updownarrow$
volatile	$f_{\text{IvP}}(x_c, x_s, x_t)$	fast

Table 6.3: Relationship between functions with respect to stability and build time.

The bathymetry data is assumed to be stable during the course of an operation. Thus the piecewise representation of this data, $\texttt{bathy}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$, is caculated once, off-line, and its creation is not subjected to real-time constraints. The function $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ is stable as long as the destination and operating depth remain constant. Our implementation of $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ in Appendix A could perhaps be run sufficiently fast if dynamic replacement were needed. Storing previously calculated versions of $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ for different depths or destinations is another viable option.

The volatile function, $f_{\text{IvP}}(x_c, x_s, x_t)$, can be calculated very quickly since so much of the work is contained in the underlying $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ function. The relationship between these three functions

results in the *appearance* that ownship is performing "dynamic replanning" in cases where the shortest path becomes blocked by another vessel. The result is a behavior that has a strong "reactive" aspect because it explicitly states all its preferred alternatives to its most preferred action. It also has a strong "planning" aspect since its action choices are based on a sequence of perhaps many actions.

6.4 Behavior 3: Quickest Path

In transiting from one place to another as quickly as possible, proceeding on the shortest path may not always result in the quickest path. If the shortest path is indeed available at all times to the vehicle, at the vehicle's top speed, then the shortest path will indeed be the quickest. Other issues, such as collision avoidance with other moving vehicles, may create situations where the vehicle may need to leave the shortest path to arrive at its destination in the shortest time possible.

Consider a situation in which a vehicle detects an obstacle moving in front, across its path. A reasonable collision-avoidance action would be to simply halt, letting the vehicle pass before proceeding. Such an action would also be a reasonable shortest-path action, but would almost certainly be sub-optimal with respect to arriving as early as possible. The quickest-path behavior simply rates actions higher with greater speed as shown below in Figure 6.14. The function created

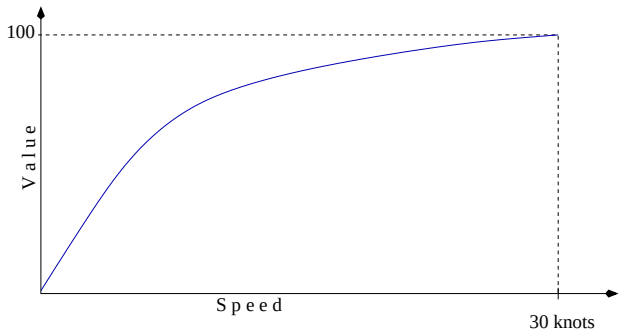


Figure 6.14: The relationship between speed and utility for the quickest-path behavior.

by this behavior, $f_{\text{IvP}}(x_c, x_s, x_t)$, or simply $f_{\text{IvP}}(x_s)$, ignores all values of course and time. However, without this payoff for speed, we will have a vehicle that is the equivalent of a person frozen at the roadside, waiting for any trace of a vehicle to vanish before crossing.

6.5 Behavior 4: Boldest Path

Sometimes there is just no good decision or action to take. But this doesn't mean that some are not still better than others. By including time, x_t, as a component of our action space, we leave open the possibility for a form of procrastination, or self-delusion. If the vehicle's situation is doomed to be less than favourable an hour into the future, *no matter what*, actions that have a time component of

only a minute appear to be relatively good. By narrowing the window into the future, it is difficult to distinguish which initial actions may actually lead to a minimal amount of damage in the future.

The boldest-path behavior therefore gives extra rating to actions that have a longer duration, i.e., higher values of x_t, as shown below in Figure 6.15. This is not to say that choosing an action

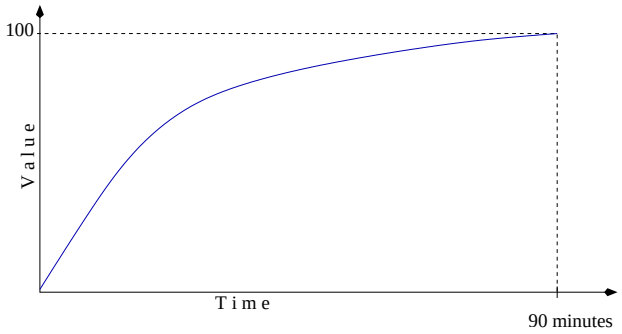


Figure 6.15: The relationship between time and utility for the boldest-path behavior.

of brief duration, followed by different one, can sometimes be advantageous.

6.6 Behavior 5: Steadiest Path

Although we seek the optimum $\langle x_c, x_s, x_t \rangle$ at each iteration of the vehicle control loop, there is a certain utility in maintaining the vehicle's current course and speed. In practice, when ownship is turning or accelerating, it not only makes noise, but also destabilizes its sensors for a period, making changes in a contact's solution harder to detect. The steady-path behavior implements this preference to keeping a steady course and speed by adding an objective function ranking values of x_c and x_s higher when closer to ownship's current course and speed.

This behavior could be split into two, one for maintaining a steady speed, and one for maintaining a steady course. Each behavior would output a one-dimensional objective function such as the one for course, shown in Figure 6.16 below. This may be a preferable implementation to separate the

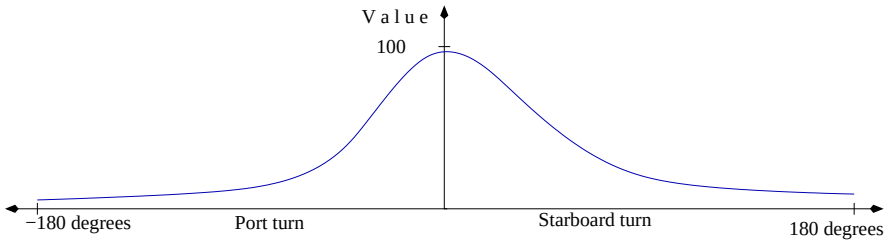


Figure 6.16: The relationship between course-change and utility for the steady-path behavior.

two issues, when their priorities are different. Note that all course changes and speed changes are legal/feasible since we do not specify the rates of change. By imposing limits on course and speed change, based on vehicle dynamics as in Fox et al. (1997), the action space could be reduced. But

in the action space we implement here, any speed or course change is feasible within the 90-minute window provided by the x_t domain.

6.7 Discussion

In this chapter, five vehicle behaviors were described, each in terms of a common vehicle model. We describe in the next chapter how these behaviors are coordinated together to control a vehicle with multiple simultaneous objectives. The description of each behavior involved characterizing the behavior's objective in terms of an objective function defined over the vehicle action space given certain information about the environment and vehicle positions. The output of each behavior is an IvP function that sufficiently approximates the behavior's underlying objective function. The construction of each IvP function involved decisions regarding tradeoffs between piece-count, accuracy, and construction time. In the next chapter we demonstrate that these tradeoffs satisfy our concerns about real-time requirements, and overall vehicle behavior.

Chapter 7

Results: Coordinating the Five Vehicle Behaviors

In this chapter we use a vehicle simulator to experiment with the five behaviors described in Chapter 6. The primary interest is in knowing if the IvP functions, from each behavior, together result in the intended overall vehicle behavior, and whether the IvP *problem*, created in each pass of the control loop (Figure 6.1(b)), can be solved quick enough to satisfy the control loop requirement.

7.1 The IvP Vehicle Simulator

The IvP vehicle simulator was built to interactively test sequences of vehicle decisions given different starting scenarios, and behavior priority weights. A snapshot of the simulator is shown below in Figure 7.1. In the main screen, the position, course, and speed of ownship and a single contact can be altered. The backdrop shows a particular destination and corresponding shortest path according to a particular $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ function, as described previously in Section 6.3.2. The shortest path to the destination is shown through a set of way-points. The contact's course and speed vector is indicated with an appropriate dashed-arrow in the figure. (Cropped snapshots from this simulator have been used in many of the figures previously shown in Chapter 6.)

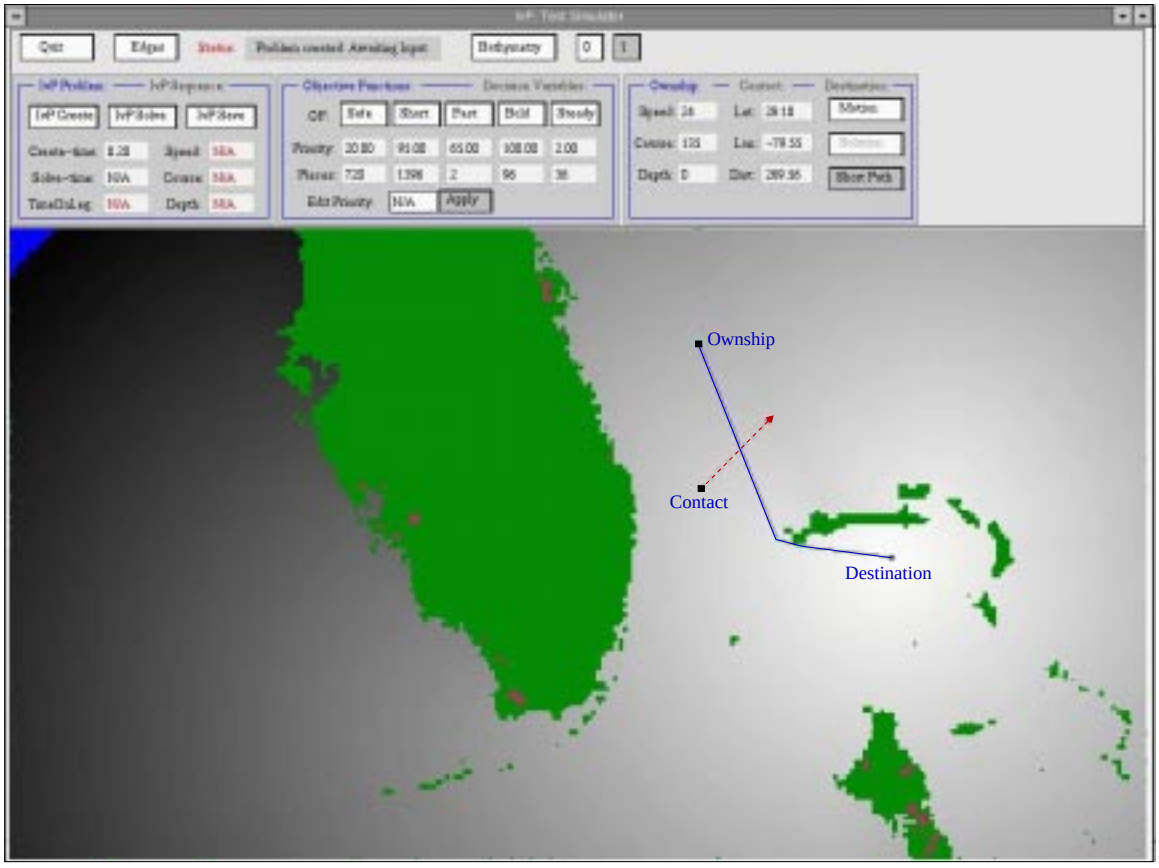
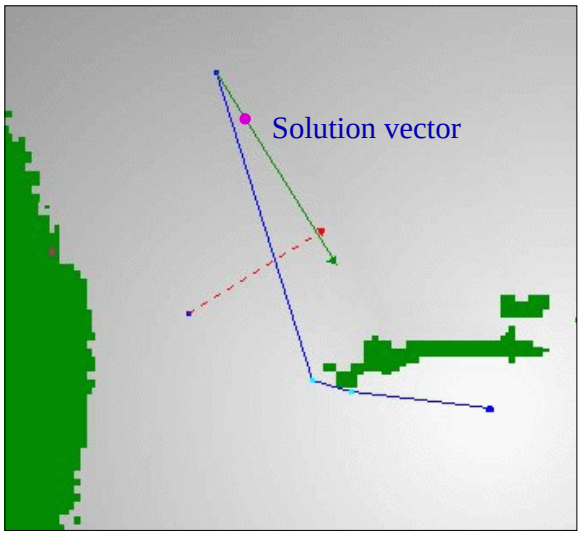


Figure 7.1: Screen snapshot of the IvP vehicle simulator.

In the upper portion of the display, there are three sub-windows. The left-hand window launches the creation of IvP functions in each behavior, and then solves the resulting IvP problem, showing the chosen values for $\langle x_c, x_s, x_t \rangle$, and the total times for problem creation, and problem solution. The middle window allows for adjustment of each behavior priority. Individual IvP functions can be viewed, and the number of pieces used in each is also displayed. In the right-hand window, the numerical values of ownship and contact's position, course and speed are displayed. From this sub-window, one may create a new $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$ function based on either a different destination, a different operating depth, or both.

7.2 Solving a Single IvP Action Selection Problem

Once the initial conditions are set, by choosing the ownship and contact positions and initial course and speed, as well as a maximum operating depth and ownship destination, each of the five behaviors have what they need to create their IvP functions. The simulator is used to examine the results within a particular iteration of the control loop. In Figure 7.2 below, an initial set of starting conditions is depicted, and the resulting solution vector is shown. The length and angle of the vector indicate



Decision variable	result
x_c	148 degrees
x_s	29 knots
x_t	35 minutes

Figure 7.2: Resulting solution vector to a single IvP problem instance.

the values for x_c and x_s in the table on the right. The dot drawn on the solution vector indicates the *intended* latitude-longitude position after ownship proceeds along this course and speed for time indicated by x_t. This solution vector represents the first maneuver in the sequence that eventually brings ownship to its destination, as shown later in Figure 7.3.

The time to create each of the five IvP functions, the number of pieces used in each function, and the total time for solving the IvP problem is given below in Table 7.1. All times are given in

Behavior function	Priority	IvP Pieces	Creation Time	Solution Time
`Safe-`$f_{\text{IvP}}(x_c, x_s, x_t)$	20	720	0.07	0.54
`Short-`$f_{\text{IvP}}(x_c, x_s, x_t)$	95	1398	0.10	
`Quick-`$f_{\text{IvP}}(x_c, x_s, x_t)$	65	4	0.01	
`Bold-`$f_{\text{IvP}}(x_c, x_s, x_t)$	100	96	0.01	
`Steady-`$f_{\text{IvP}}(x_c, x_s, x_t)$	2	36	0.01	

Table 7.1: A break-down of the created and solved IvP problem.

seconds, rounded up to the next 1/100 second. Individual IvP problems can be saved to a file, and solved off-line by a brute-force algorithm to check for correctness. The solution to this particular problem instance can be verified by brute force in 50.78 seconds, roughly two orders of magnitude slower than using the IvP algorithm. Both were run on a 1.1 GHz Pentium III with 512 Megabytes of RAM.

7.3 Solving a Series of IvP Action Selection Problems

In testing a series of action selection problems, the simulator begins with the starting conditions described above in Section 7.2, and repeatedly updates the position of ownship and the contact, at fixed time intervals. Thus smaller gaps between markers indicate slower vehicle speed, as between markers 6 and 10 below in Figure 7.3. Each time an action is executed, it is held for 10 simulated

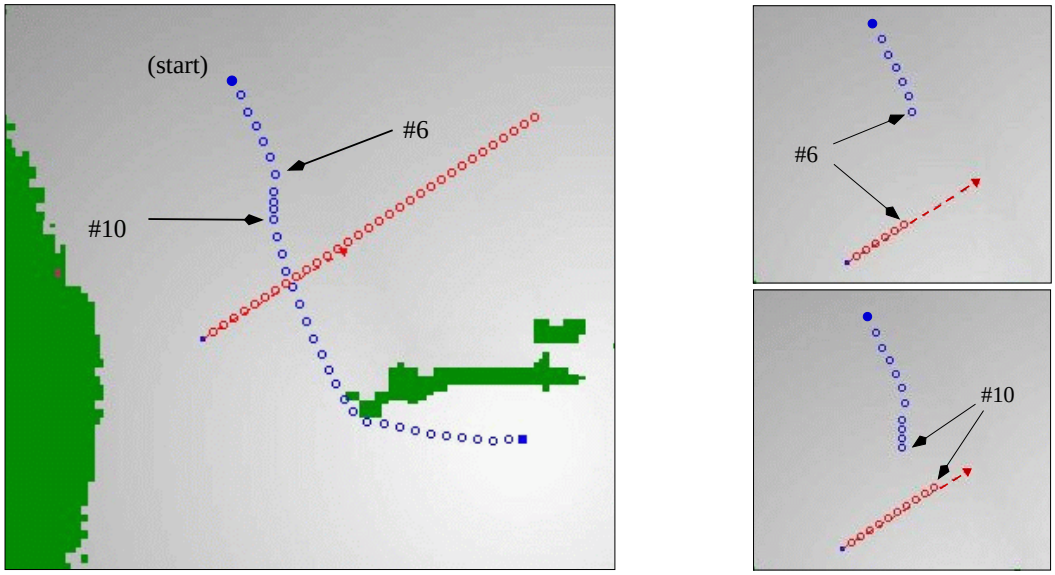


Figure 7.3: Ownship slows down and cuts behind the moving contact.

minutes. This defines a loose control loop requirement of creating and solving an IvP problem in less than 10 real minutes. For the objective functions created and used here, the control loop can be executed once per second. The choice of 10 minutes of simulated time per step is made for convenience in viewing the simulated results.

The scenario in Figure 7.3 above depicts a two-part evasive maneuver by ownship while transiting to its destination with the contact moving across its bow. The first part occurs at steps 6 and 7, when ownship slows down and begins to direct its course to intersect the contact trajectory at a point well *behind* the contact. Prior to step 6, the intersect point had been in front of the contact. At step 10, when ownship is sufficiently behind the contact, and with the closest point of approach apparently in the past, ownship begins to speed up again, and adjust its course toward its destination.

We can see the effects of a difficult, i.e., conflicted, decision on solution time, if we plot the observed solution time at each step, for each IvP problem, as below in Figure 7.4. The decision at steps 6 and 7, to slow down and change course, is clearly at odds with the shortest-path and quickest-path behaviors, and reflects the emerging dominance of the safest-path behavior. When relatively un-conflicted, as in steps 12 and beyond, the solution time remains at about 0.25 seconds.

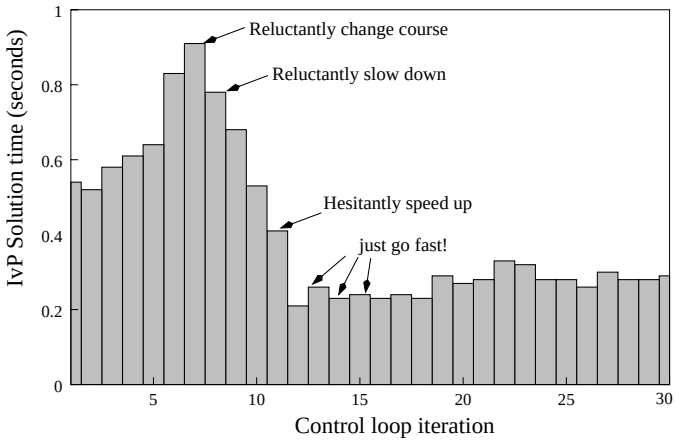


Figure 7.4: IvP solution time for each control loop iteration.

7.4 Scenarios with Moving, Maneuvering Contacts

The four scenarios presented below demonstrate the ability of the vehicle, governed by our five behaviors, to react to changes in motion in another moving obstacle or vehicle. In the last three scenarios, the contact is programmed to make a single course or speed change at some pre-determined time step. All four scenarios start with the same initial conditions and behavior weights. In the first scenario the contact does not maneuver and stays on a simple linear track. This scenario is provided for comparison in the following three scenarios where the contact does indeed change course or speed. In this simulator, knowledge of a change in the contact trajectory is immediately available to ownship and its behaviors. In reality, this information would be available only after some delay, and with some uncertainty.

Scenario 1 The scenario in Figure 7.5 below is used as a baseline for comparison with the following three scenarios. In this case the contact remains at a steady course and speed of 45° and 20 knots. The priorities for each of the five behaviors are: [Safe:20, Short:95, Quick:65, Bold:100, Steady:2].

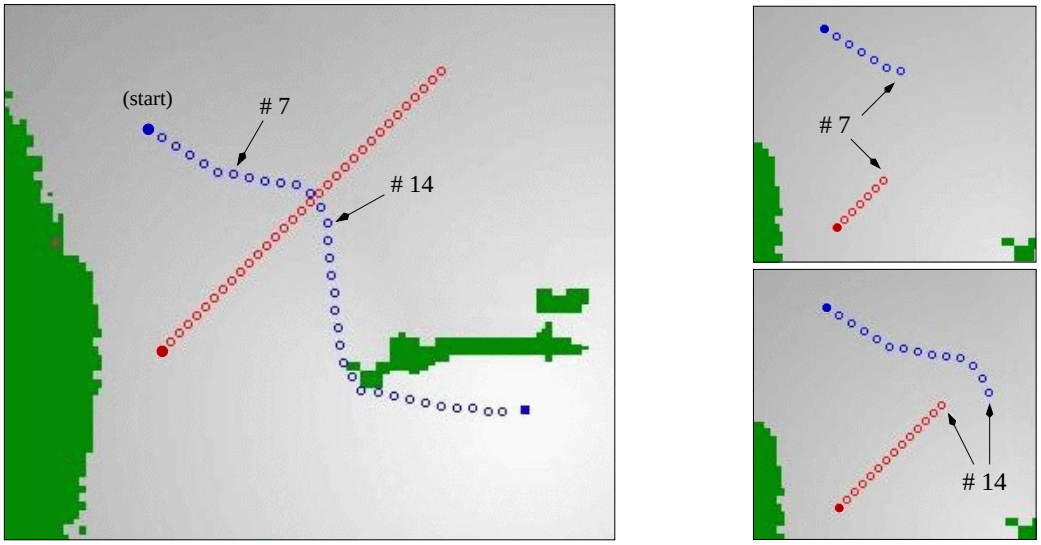


Figure 7.5: Ownship uses its speed advantage to cross the bow of contact.

With the contact remaining at 20 knots, ownship has a 10 knot speed advantage due to its maximum speed of 30 knots. Here ownship uses this speed advantage to cross in front of the contact before maneuvering back toward the destination. Steps 7 and 14 are identified in the figure as roughly the points where these two stages begin. In this scenario, ownship remains at its top speed at each step since the top speed initially contributes to a larger distance between the two vehicles as ownship crosses in front of the contact, and even with the contact safely behind and moving away from ownship, the higher ownship speed contributes to an earlier arrival time at the destination.

Scenario 2 In Figure 7.6 below, the scenario is identical to the one above in time steps 0 thru 4. After step 4, the contact changes its course from 45° to 90°. After this course change, it is no longer feasible for ownship to maneuver in front of the contact as it did in Figure 7.5.

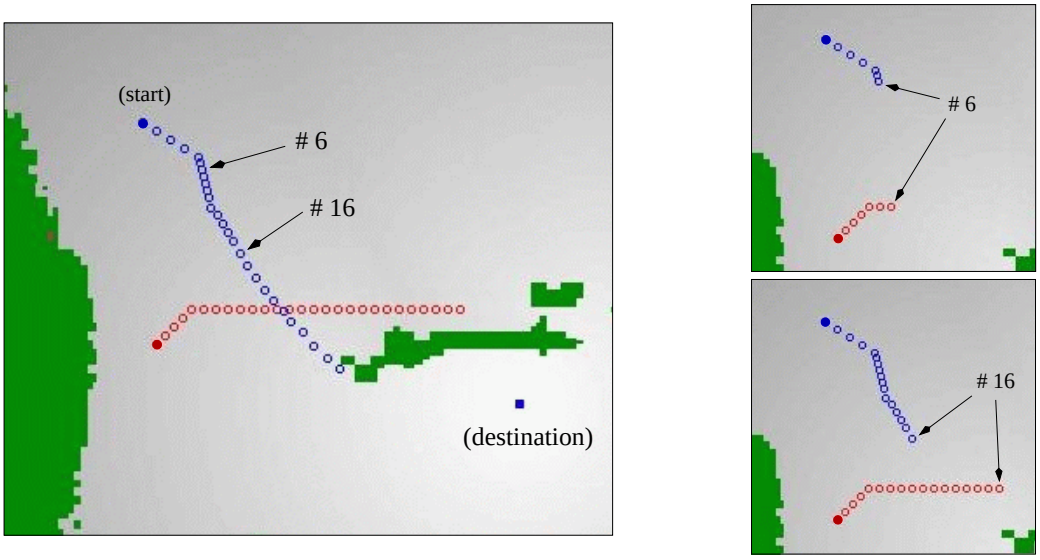


Figure 7.6: Ownship reacts to the contact's course change by slowing and cutting behind the contact.

Ownship instead slows and changes its course to cross behind the contact on the way to the destination. The priorities of the behaviors are the same as in the previous scenario. Note that reaction to the contact's course change is immediate in that ownship's decision making process between steps 4 and 5 already reflects knowledge of the new contact course. After time step 16, when ownship is sufficiently behind the contact, ownship begins to speed up again. The chosen speeds between steps 6 and 16 reflect the influence of the safest-path behavior, which prefers larger distances between the two vehicles.

Scenario 3 In this scenario, depicted in Figure 7.7 below, the starting conditions are the same as in the previous two scenarios. This scenario differs from the previous one only in that the contact makes its course change after step 9, instead of step 4.

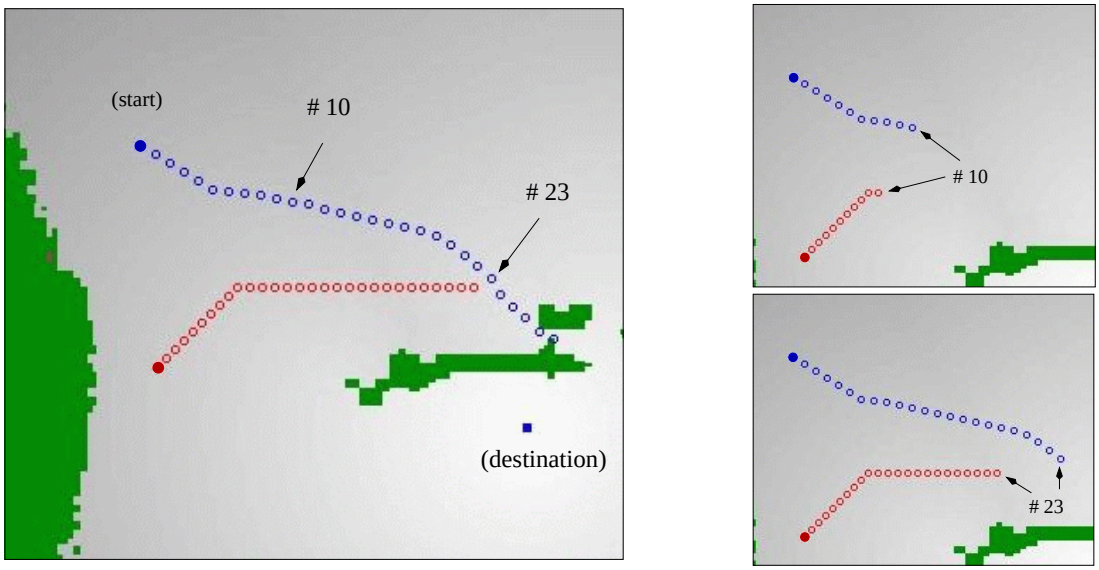


Figure 7.7: Ownship reacts to the contact's course change and adjusts its path to the destination.

At this point when the contact changes its course, ownship opts not to slow and drop behind the contact as in the previous scenario. Instead, it maintains its course, at top speed, and proceeds to the destination around the eastern side of the island. Although slowing and dropping behind the contact would indeed result in a shorter path to the destination, the need to slow down for 8 or 10 steps may result in a longer overall time to transit to the destination. At the critical decision point, after step 10, ownship is at a point where the path to the destination around the eastern side of the island is *nearly* as short as the path around the western side. The influence of the fastest-path behavior is enough at this point to tip the scales to the slightly longer path around the eastern side.

Scenario 4 In this scenario, we demonstrate how ownship reacts to a change of speed in the contact trajectory. The contact changes speed, after step 4, from 20 knots to 25 knots. The initial conditions are the same as in the previous three examples. The increase in contact speed makes it difficult for ownship to race in front of the contact as it did in the first scenario.

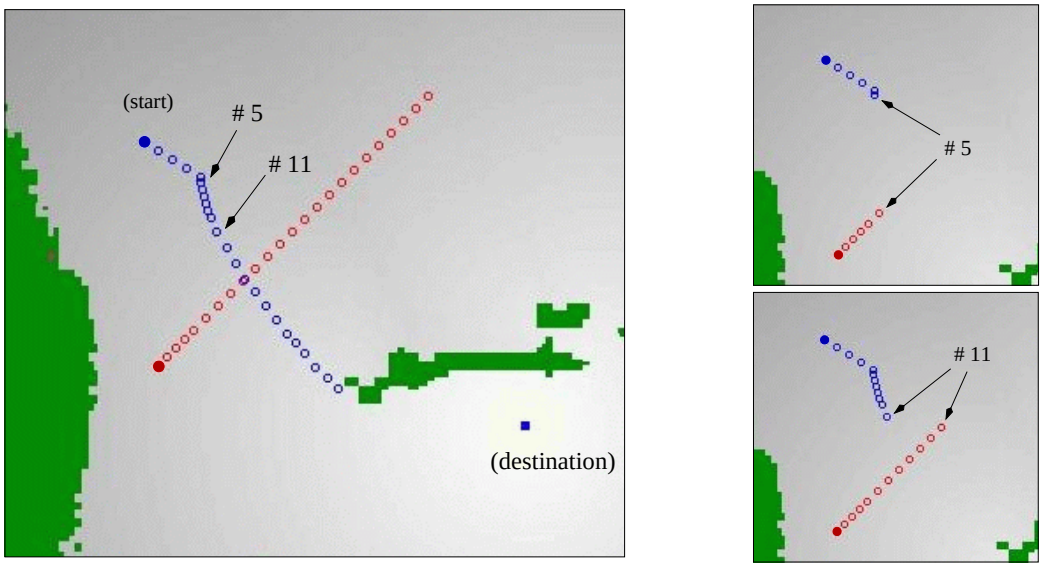


Figure 7.8: Ownship reacts to the contact's speed change by slowing and cutting behind the contact.

After step 4, when the contact increases speed, ownship abandons the strategy of crossing the contact's bow, and instead decreases speed and takes a course cutting behind the contact. Once ownship is sufficiently behind the contact, the urge to speed up (from the fastest-path behavior) is less conflicting with the safest-path behavior resulting in an increase in speed after step 10.

7.5 Discussion

The primary aim of this chapter was to verify that behaviors could produce IvP functions that were sufficiently accurate with sufficient speed, and that the resulting IvP problem could then be solved quickly enough to satisfy realistic time constraints. A control loop iteration once every second was deemed sufficiently fast here. In the problem that originally motivated this work, a submarine maneuver decision aid, (Benjamin et al., 1993), a decision computed in less than a minute was deemed sufficient. It is expected that tactical decisions will be made much more frequently on AUVs in current development.

A second aim of this chapter was to demonstrate a particular approach to motion planning with moving contacts, composed by not only the collection of behaviors described in the previous chapter, but also the multi-objective optimization approach to action selection in general. We were motivated by systems emphasizing the ability to react in the presence of moving obstacles that frequently change

trajectories, e.g., Bruce and Veloso (2002). In Bruce and Veloso (2002), the domain is RoboCup, where multiple moving obstacles are considered and decisions are made 30 times per second. Each obstacle, however, is treated as a stationary obstacle that happens to (perhaps) be in a different spot as time progresses. By reasoning about the perceived trajectory of the obstacles, decisions can likely be made earlier that preclude unwanted relative positions. Thus we were also motivated by works that explicitly reason about time and obstacle trajectories, such as Kindel et al. (2000). In Kindel et al. (2000), collision-free trajectories are calculated fairly quickly given one or more moving obstacles. In the underwater domain, however, concern about relative position is not restricted to the extreme case of collision avoidance. We were also therefore motivated by multi-objective optimization approaches to action selection such as the work by Rosenblatt (1997) and Pirjanian (1998). In these works, however, no techniques suitable for large action spaces were presented.

Chapter 8

Conclusions and Future Work

8.1 Conclusions and Contributions

In this work we presented the interval programming model for representing and optimizing multiple objective functions. The motivation for this model was to provide a multi-objective optimization method with a unique balance of sufficient expressive power and flexibility, and solution algorithms that are tractible for the applications of interest to us, namely autonomous vehicle control. Prior to this work, existing multi-objective action selection methods were limited to explicit evaluation of elements in the decision space. We discussed the motivation for overcoming this limitation in Section 2.2, and demonstrated the payoff for doing so in Chapters 6 and 7.

The main contribution of this work is the IvP model itself. Piecewise functions are not new, but using them to represent multiple objective functions, and using the set of combinations of pieces from each function as the solution space, is indeed a new approach. We provided a set of algorithms that solved sufficiently large problems, representing sufficiently realistic vehicle control scenarios, with sufficient solution speed in practice. The application of branch and bound to this problem, and the corresponding bounding algorithms, are another contribution of this work.

In the IvP model, piece boundaries were defined primarly as intervals over decision variables, i.e., rectilinear pieces. A significant extension to this was the use of intervals over certain functions of decision variables. Generally speaking, this allowed IvP functions to be more accurate in representing an underlying function, while also using less pieces. More importantly, in our implementation of vehicle control presented in Chapter 6, the IvP functions using these non-rectilinear pieces represented an objective function defined over action consequences, rather than actions. This provided us with the important ability to coordinate behaviors that reason at different levels of abstraction.

We did not claim to make a contribution to the body of existing algorithms for approximating functions with piecewise linear functions. We did, however, implement a relatively unsophisticated bag of tricks to produce the objective functions used in our vehicle simulator. We needed to show

that this phase of the control loop would not be a time bottleneck. We expect that when more mature methods are applied, and behaviors produce their IvP functions in parallel, this phase will consume even less of the total control loop time.

Finally, we have presented a vehicle simulator, implemented in C++, for testing the combined effect of different behaviors vying for vehicle control. In this simulator, the behavior characteristics and priorities can be altered and re-run on saved scenarios. We used the results in this simulator to confirm that IvP problems can be built and solved with the necessary speed and accuracy for controlling an autonomous vehicle under the conditions described.

We hope that the IvP model provides a viable, easy-to-use technique for those wishing to solve multi-objective action selection problems in autonomous vehicle control.

8.2 Future Considerations

In this work we have provided a model for representing and solving multi-objective action selection problems in autonomous vehicle control. We look forward to moving this work from simulation onto actual vehicles. As this happens, there are other things we will introduce first to the simulation environment, and other issues concerning the general IvP model that need to be better understood and improved upon.

The interval programming model issues

The complexity of the branch and bound algorithm described in Chapter 5 is primarly tied to the number of nodes in the search tree, formed by overlapping pieces from each function. With no prior knowledge of piece layout, it is possible that each piece from a particular function may intersect with any given piece from another function. This results in a search tree that grows exponentially with the number of pieces. With an equal amount of variables and objective functions, contrived examples can be built where this worst case is indeed the case. In practice, however, with the functions typically found, the result is a significantly reduced search tree. It should be possible to find a tighter bound on the size of the search tree under "normal" circumstances. Conditions for such circumstances need to be identified, and better bounds understood.

Vehicle Simulation

One of the first additions to the simulation environment will be to fold in more realistic circumstances in which the controlled vehicle obtains information about the environment. Up to now we have assumed the vehicle precisely knows the positions and trajectories of other vehicles. In reality, this information is acquired incrementally and with a degree of uncertainty. The actions chosen by the controlled vehicle have an impact on the quality of the acquired information. We expect to connect to the techniques used in concurrent mapping and localization (CML) to build behaviors that guide a vehicle to collecting better information. We want a vehicle that simultaneously makes progress on the CML problem while performing other vehicle behaviors. And we want a vehicle that is able to do both while shifting priorities as the needs of each problem change due to circumstances or level

of progress.

Another imminent change to the simulation environment will be the addition of other vehicles. In our scenarios, presented in Chapter 7, there was one ownship under IvP -control and one contact under scripted-control. There are many interesting variations of this to consider. We would also like to give vehicles joy-stick control to offer a more intelligent adversary to ownship, and to compare performance of human control of ownship against IvP -control. The addition of multiple contacts will create more difficult navigation challenges to ownship, and IvP problems with more objective functions. We also intend to experiment with multiple ownships to simulate group cooperative behavior, such as formation following, and swarming.

With respect to our simulated vehicle model, we intend to augment this model to include decisions about depth, acceleration, rudder angle, and plane angle. In the current simulation model, the vehicle makes course and speed changes with an assumed rudder angle and acceleration. In reality, there may be a dozen or so choices for each, and such choices can have a significant effect on self-noise and sensor stability.

Transitions

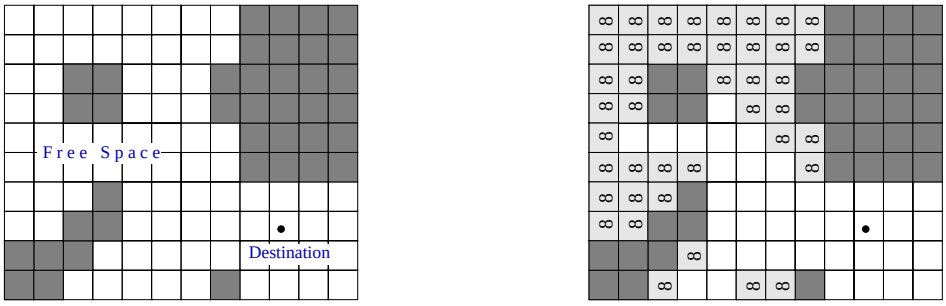
There are three existing Navy projects that IvP is in various stages of transition. The first has nothing to do with autonomous vehicle control, but addresses the need for multi-objective optimization in the design of complex systems. This work is supported by Dr. Kam Ng at ONR, under the Undersea Weapons Design & Optimization (UWD&O) effort. In designing a complex system, such as a torpedo, there are competing design objectives regarding issues of speed, stealth, cost, size etc., that need to be resolved in the ultimate design. The IvP model is being applied to represent such problems, and support a design environment where the designer can interactively vary the priorities of design objectives and see the effect in the design space. In the IvP model, when an objective function priority changes, a new scalar weight is applied to the linear interior functions of each piece. The work of building an IvP function, to represent an underlying design objective, therefore does not change as the user interacts by applying new priority weights.

The Navy is also building an unmanned undersea vehicle (UUV), at the Naval Undersea Warfare Center (NUWC) in Newport RI, called Manta. A different group at NUWC has been tasked to build an unmanned surface vehicle (USV), called Spartan. Both of these vehicles need to operate autonomously in the presence of other moving vehicles in a potentially adversarial environment. Due to their complex environment and missions, these two vehicles should be well-suited to the application of multi-objective action selection.

Appendix A

Creating a Piecewise All-Sources Shortest Path Function

In this appendix we provide the algorithm for building the function $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ described in Section 6.3.2. In building $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ for a particular destination and depth, the latitude-longitude space is divided into either free space, or obstacles, based on the $\texttt{bathy}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ function. A simple case is shown below in Figure A.1(a). In the first stage of building $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$, all lat-lon pieces



(a) Given destination and free space

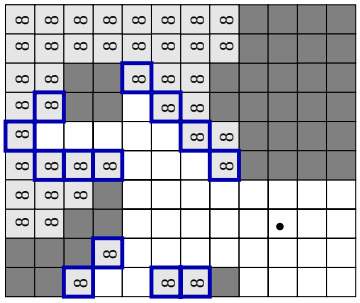
(b) Direct-path pieces

Figure A.1: Identifying direct-path pieces in the initial stage of building $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$.

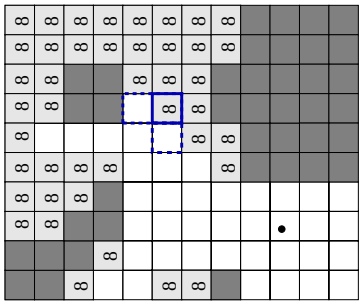
are identified such that all interior positions of the piece are reachable to the destination on a single direct linear path. In Figure A.1(b), these are indicated by the white empty pieces. The other pieces are marked with ∞, since their distance to the destination is initially unknown. Choosing these pieces to be uniform was done only for clarity in these examples. The pieces in $\texttt{spath}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ and $\texttt{bathy}(\texttt{p}_{\text{LAT}}, \texttt{p}_{\text{LON}})$ used in results of Chapter 7 were not uniform, and the algorithm provide below is not dependent on uniform pieces.

After the first stage, there exists a "frontier" of pieces, each having a directly-reachable neighbor

that has a known shortest-path distance. These frontier pieces are shown below in Figure A.2(a). For such pieces, one can at least improve the "∞" distance by proceeding through its neighbor. But



(a) All frontier pieces



(b) A frontier piece with two neighbors

Figure A.2: Identifying the frontier in intermediate stages of building $\texttt{spath}(\texttt{p}_{\texttt{LAT}}, \texttt{p}_{\texttt{LON}})$.

consider the case in Figure A.2(b), where a frontier piece has two such neighbors. Unless an effort is made to properly "orient" the frontier, unintended consequences may occur. Furthermore, even if the correct neighbor is chosen, we can often do better than simply proceeding through the neighbor.

We describe now our implementation of an all-sources shortest path algorithm. The only value we ultimately care about for each piece is the linear interior function indicating the shortest-path distance for a given interior position. However, the following intermediate terms are useful:

$$\begin{aligned} \texttt{dist}(\texttt{pc}_a, \texttt{pc}_b) = &\quad \text{Distance between center points of } \texttt{pc}_a \text{and } \texttt{pc}_b. \\ \texttt{pc}_a{\rightarrow}\texttt{dist} = &\quad \text{Distance from the center point of } \texttt{pc}_a \text{to the destination.} \\ \texttt{pc}_a{\rightarrow}\texttt{waypt} = &\quad \text{The next waypoint for all points in } \texttt{pc}_a. \end{aligned}$$

After the first stage of finding all directly reachable pieces, the value of $\texttt{pc}_a{\rightarrow}\texttt{waypt}$ for such pieces is simply the destination point, $\langle \texttt{d}_{\texttt{LAT}}, \texttt{d}_{\texttt{LON}} \rangle$, and `NULL` for all other pieces. By keeping the waypoint for each piece, we can reconstruct the actual path that the shortest-path distance is based upon. The basic algorithm is given in Figure A.3. Three subroutine calls are left un-expanded: `setDirectPieces()`, `sampleFrontier()`, and `refine()`, on lines 0, 3, and 5. Explanations for the latter two will be given without code, as was done for the first one. The basic idea of the while-loop is to continue refining pieces on the frontier until a set amount (in this case 100) of successive refinements fail to exceed a fixed threshold of improvement.

The function `sampleFrontier(amt)` searches for pairs of neighboring pieces, $\langle \texttt{pc}_a, \texttt{pc}_b \rangle$, where one piece could improve its path by simply proceeding through its neighbor. The pairs of pieces are randomly chosen by picking points in the latitude-longitude space. The opportunity for improving $\texttt{pc}_a$ through its neighbor, $\texttt{pc}_b$, is measured by: $\texttt{opp}_\texttt{a} = \texttt{pc}_a{\rightarrow}\texttt{dist} - (\texttt{dist}(\texttt{pc}_a, \texttt{pc}_b) + \texttt{pc}_b{\rightarrow}\texttt{dist})$. Each pair is then placed in a fixed-length priority queue, where the maximum element is a (frontier)

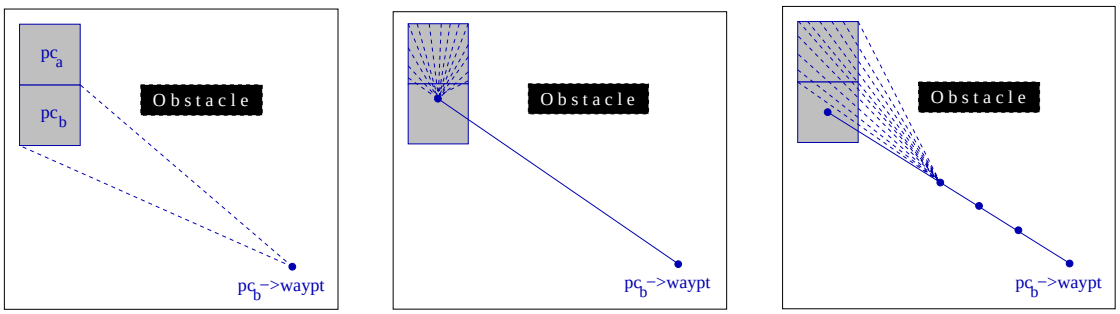
```
All-Pairs Shortest Path()
0. setDirectPieces()
1. threshCount = 0
2. while(threshCount < 100)
3.    sampleFrontier(50)
4.    pqueue→extract-max(pc_a, pc_b)
5.    val = refine(pc_a, pc_b)
6.    if(val < thresh)
7.       threshCount = threshCount + 1
8.    else
9.       threshCount = 0
```

Figure A.3: The all-sources shortest path algorithm.

pair with the greatest opportunity for improvement. This queue will never be empty, but will eventually contain only pairs with little or no opportunity for improvement. There is also no guarantee that the same pair is not in the queue twice.

After a certain amount of sampling is done, the maximum pair is popped from the queue as in line 4 in Figure A.3. The function `refine(`$\texttt{pc}_a$`,` $\texttt{pc}_b$`)` is then executed, returning the measure of improvement given by `val`. The counter, `threshCount`, is incremented if the improvement is insignificant, eventually triggering the exit from the while-loop. If the improvement in $\texttt{pc}_a$ *is* significant, it will likely create a good opportunity for improvement in other neighbors of $\texttt{pc}_a$. These neighbors (pairs) are therefore evaluated and pushed into the priority queue.

The execution of `refine(`$\texttt{pc}_a$`,` $\texttt{pc}_b$`)` should, at the very least, make the simple improvement depicted below in Figure A.4(b), where $\texttt{pc}_a$→`waypt` is set to an interior point in $\texttt{pc}_b$, e.g. the center



(a) Opportunity for improvement (b) Simplest improvement (c) Better improvement

Figure A.4: Refining the shortest-path distance with help from a neighbor.

point, and the linear function inside $\texttt{pc}_a$ is set to represent the distance to this new way-point, plus the distance from that way-point to the destination. Refinements such as the one depicted in Figure A.4(c) search for shortcuts points along the path from $\texttt{pc}_b$ to its way-point. If such a point is found, it becomes the value of $\texttt{pc}_a$→`waypt`, and the appropriate linear interior distance function is calculated. The value returned by `refine(`$\texttt{pc}_a$`,` $\texttt{pc}_b$`)` is the difference in $\texttt{pc}_a$→`dist` before and after the function call.

Appendix B

Case Studies of Three Action Selection Methods

In this appendix three applications are examined, each using multifusion action selection, that make different design choices based, in part, on a tractibility vs. correctness tradeoff, and in part on the particular application requirements. These three applications, arguably, represent the current state-of-the-art in action selection methods for autonomous agent control. We do not address any of the non-multifusion methods since a strong case has already been made for their inadequacy in the applications of interest to us.

B.1 Voting methods

In Rosenblatt (1997), the application of interest was autonomous land vehicle control. The work[1] was tested on real vehicles travelling on real roads. The primary decision variables were turn angle, speed, and camera field-of-regard (composed of camera pan and tilt). The domain for the turn radius contained 51 different elements with the ranges shown in Figure B.1 with 0.005 increments. The field-of-regard domain had only five different pan values, while the tilt angle was kept constant (pg. 70). The speed domain was continuous, which sounds inconsistent with the idea of voting. The speed was simply set to the maximum allowable speed once the turn radius was decided, based on constraints such as vehicle tipping (pg 68-69, 81).

Each behavior produced action-value pairs (weighted votes) to be simply added, in a manner as shown in Figure 2.10. This simple combining approach also implies that the "best" action is simply the one with the highest sum of votes. In Rosenblatt (1997), voting was not done in a multidimension decision space. Decisions for each variable were made sequentially with the results of the first variables constraining the assignment of later variables. It appears that the camera angle

[1] Rosenblatt (1997) dubbed his work DAMN: Distributed Architecture for Mobile Navigation

Figure B.1: Turn radius and camera field-of-regard choices (Rosenblatt, 1997, p.65, 71).

can be set in this way without scarificing performance, but for the reasons mentioned in Section 2.2.4 and Figure 2.9, the speed and turn settings should be made together. Rosenblatt (1997, p. 81, 142) himself noted the importance of an arbiter that accepts votes for combined turn and speed commands, but rejected the idea believing it to be "unwieldy".

B.2 Action Maps

In Riekki (1999), the application of interest was simulated soccer, namely the RoboCup competition. RoboCup features both real, physical robots playing soccer, as well as simulated robots. It appears that the latter was the target in this work. The primary decision variables were speed and direction of the robot. Riekki's work is singled out here because it is a rare example of action selection taking place in multi-dimensional space (as the taxonomy of Figure 2.11 indicates). The maps, produced by the behaviors, are composed of a set of action-value pairs, and can be depicted in a polar coordinate system, with the robot at the center, as shown below in Figure B.2 taken from Riekki (1999, p.52). Maps for reaching an object have strictly positive weights and maps for avoiding objects have strictly

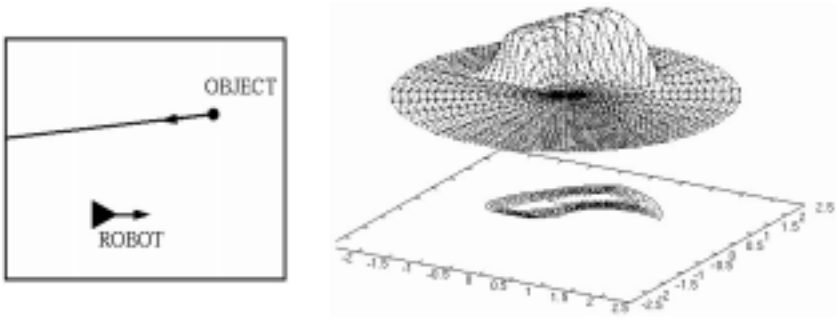
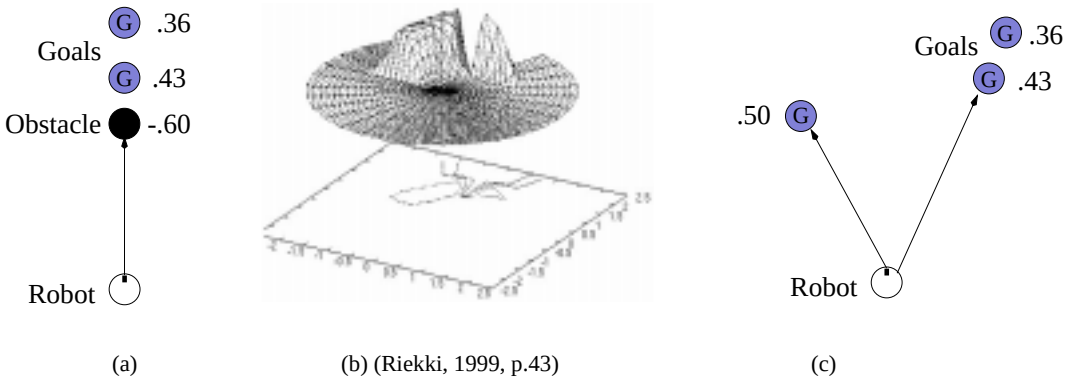


Figure B.2: An action map to maneuver a robot *to* an object (Riekki, 1999, p. 52).

negative weights. Action selection is done by successively overlaying each map by combining each point in the decision space and finally taking the action with the highest combined value.

There are two interesting details relevant to our discussion. The first is that actions are *not* combined by additive voting. Instead, the weight with the highest absolute value from each individual

map is the final value given to the action (Riekki, 1999, p. 56). The reason for this is that the magnitude of the weights is directly proportional to the distance between an object and a robot. By taking the largest absolute value, this ensures that an action will not be chosen that moves the robot through an obstacle, regardless of the amount of additive rewards that may lie behind it. In Figure B.3 (a) we show the situation where two objects to be reached are lined up behind an obstacle. By



(a) (b) (Riekki, 1999, p.43) (c)

Figure B.3: Combining actions. (Riekki, 1999, p.43)

taking the maximum absolute value, the map in (b) is obtained, which still may allow actions that take the robot *near* the goals. This stategy appears to work in this case, but consider (c) where adding the weights seems more appropriate. This situation less than desirable in that, in general, it is best if the correctness of that action selection method is independent of the world situation.

The interesting point is that the action maps in Riekki (1999) were combined in a predefined order, and that if a particular action reached a threshold condition, then the action selection method would terminate prematurely without considering the remaining action maps (pg. 43). This may be an indication of the need to speed up the action selection process since decisions were made in the RoboCup Soccer Server once every 100 milliseconds. As we mentioned in Section 2.2.5, methods that explicitly evaluate each action in a high dimension action space (Riekki, 1999, p.56) are likely to be challenged in meeting time requirements. However, this ASM, which is one of the more advanced in terms of the speed/complexity taxonomy given in Section 2.2, may also resemble the most primitive in the same taxonomy if it is in fact ignoring the outputs of some behaviors for the sake of making quicker decisions.

B.3 Fuzzy methods

In Saffiotti et al. (1999), as well as Yen and Pfluger (1995), the application of interest was mobile robot navigation through an office environment. The primary decision variable was the direction of the robot. These two works are a good representation of approaches using fuzzy logic in mobile robot control, and post-date several other papers by the same authors on the subject. Their approach is described in terms of how they rate actions from individual behaviors, combine actions from different

behaviors, and then search for the single best action. Both approaches handle the first step in the same, but flawed, way. This flaw leads to two different approaches in making the last two steps perform acceptably in limited situations.

Rating actions The first step is for each behavior to create a "desirability function" mapping each possible control action to a number between 0 and 1. In Figure B.4, taken from Saffiotti et al. (1999, p. 188), two behaviors are shown with their action-ratings that, avoid obstacles and follow a hallway respectively. Each function is created by a combination of fuzzy rules with antecedents

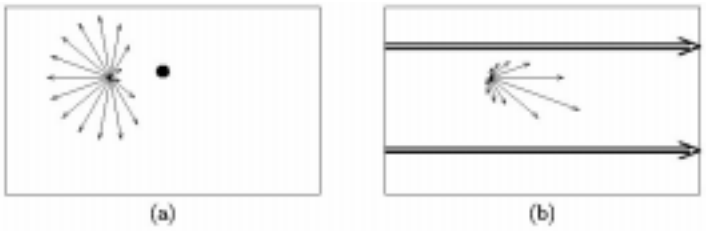


Figure B.4: Avoid object, and hallway following behaviors (Saffiotti et al., 1999, p. 188)

referring to fuzzy state descriptions. The important point to notice is that behaviors describing actions to be *avoided*, as in B.4(a) above, still map actions to the domain $[0, 1]$. This is different than both Rosenblatt (1997) and Riekki (1999), which use the domain $[-1, 0]$ for avoid-actions. This is the root of the problems found in the next two steps.

Combining actions The second step is to combine the individual values for each action taken from each behavior. In Saffiotti et al. (1999), this step is called "blending of behaviors". Instead of voting, however, actions are combined by conjunctive combination. Saffiotti et al. (1999) pointed out that this method is flawed in certain contexts, and offered a method called *context-dependent* blending, which requires some knowledge of the world-state. These first two steps, along with the third step, defuzzification, are shown in Figure B.5 from Saffiotti et al. (1999, p. 195). The use of

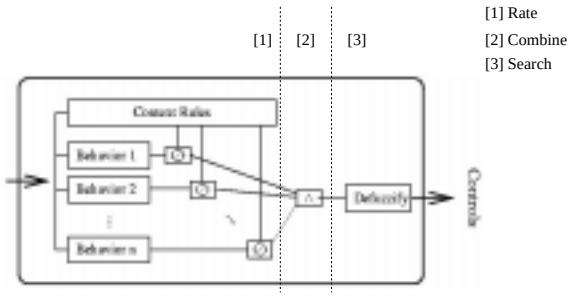


Figure B.5: Behavior combination. (Saffiotti et al., 1999, p.195)

context serves to "discount the desirability function" of each behavior, but the blending remains simple conjunctive combination (pg. 194, 5.4). It seems that "context-dependent" blending has less

to do with the blending than it does the rating of the actions (via context priorities) before the blending.

Conjunction does have the convenient property that it is well defined on values $[0, 1]$, and that behaviors that rate a particular action to be unacceptable (equivalent to a constraint), will have a zero value that is retained regardless of what values are conjoined with it. A fundamental problem with mapping everything to the $[0, 1]$ domain for the sake of using multivalued logic, is that an action rated by a behavior to be *very undesirable* (say -0.9 in $[-1, 0]$) is mapped to 0.1 in $[0, 1]$. Furthermore, an action that is rated by a behavior to be *slightly desirable* could also be rated with a 0.1 in $[0, 1]$. It is certainly *not* the case that something that slightly contributes to an objective is equivalent to being something that is very detrimental to an objective, even thought both may be mapped to the same value in $[0, 1]$. Figure B.4(a) from Saffiotti et al. (1999) would be more aptly depicted as in figure B.6, if negative weights were allowed. Once this mapping of everything

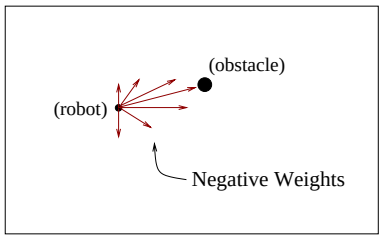


Figure B.6: An alternative "avoid object" behavior.

from $[-1, 1]$ to $[0, 1]$ is made, significant information is lost, and it is unlikely that *any* method of combination will be satisfactory in a wide variety of situations. This becomes more evident as an application has many simultaneous objectives (which Saffiotti et al. (1999) did not have). Consider five behaviors, each with a moderate rating of 0.4 to one particular action. If we were lucky enough to have one action that contributed fairly well for *five* behaviors, that this action may be preferred to an action that performs a little better (e.g. 0.5) for only one behavior. But if both positive and negative behaviors had been mapped to $[0, 1]$, we have no idea if the five 0.4 values are instead five strong reasons *not* to pick the action. In fact, ironically, the conservative operation, conjunction, is perhaps the safest thing to do given that the crucial information is lost.

Choosing an action The task of defuzzification is to find a single, "crisp" action that is to be finally chosen for execution. In both the voting and action map methods, this was simply the action with the highest ranked value, after votes were added, or maps combined. Defuzzification also begins with a combined mapping from actions to values, but doesn't simply take the action with the highest value. Several techniques have been offered such as the Mean of Maximum (MOM), Center of Area (COA) and Centroid of Largest Area (CLA). The MOM and COA methods have known flaws as pointed out in Yen and Pfluger (1995), where the CLA method was shown to work well in some cases where the COA was flawed, as shown in Figure B.7(a). In Yen and Pfluger (1995) they experiment with these three methods and report that the CLA method worked best. In Saffiotti et al. (1999),

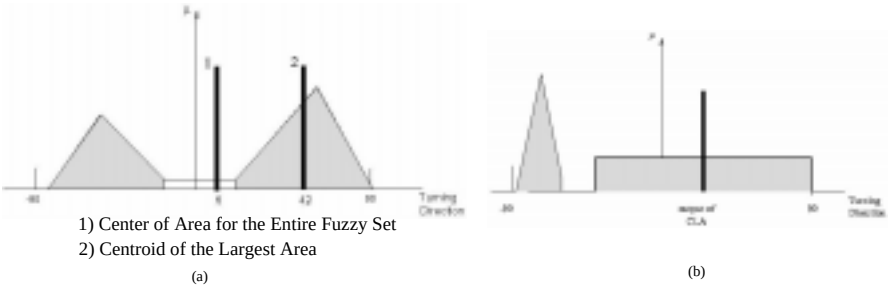


Figure B.7: Defuzzification methods. Yen and Pfluger (1995, p.14) and Pirjanian and Mataric (1999, p.6)

the COA method was used despite its known flaw when a multimodal function is to be defuzzified as in Figure B.7(a). Their "empirical stategy" (p. 190) was to prevent such functions from ever having to be defuzzified, by ensuring that two rules that, *would* together produce a bimodal function, have mutually exclusive preconditions. This seems plausible in a simple environment like the one tested, but is unlikely to scale well for two reasons. First is that it would preclude having any single behavior that generates a single bimodal function, rating elements of the decision space in a manner that reflects two genuine noncontiguous ways of achieving the same goal. Examples of this were discussed with underwater vehicles in Chapter 6. The second reason it would not scale well is that the method of context-blending in Saffiotti et al. (1999) needs to surpress behaviors, according to context, that may generate a bimodal function in the presence of a stronger conflicting behavior. This approach begins to resemble the no-compromise methods discussed in Section 2.2, which will likely lead to suboptimal actions for the reasons mentioned in that section.

Summary of fuzzy techniques In summary, no defuzzification technique has been put forth that isn't readily open to a flaw-revealing counterexample (see also Pirjanian and Mataric, 1999). We contend that such a technique will continue to be elusive as long as action-ratings are mapped into the $[0, 1]$ domain, an apparent necessity for the fuzzy logic formalism. Furthermore, in Saffiotti et al. (1999) speed was determined after a suitable direction was found, and in Yen and Pfluger (1995), there was no mention at all of speed or any other decision variables. It remains to be seen how their techniques would fare in a multidimension decison space.

Appendix C

The Integer, Nonlinear and Convex Programming Models

In this appendix, three mathematical programming models are discussed as additional background to the discussion in Chapter 2.3. For more, see Winston (1995) or Ecker and Kupferschmid (1991). The example integer programming problem presented below in Section C.1 also doubles as an example linear programming problem, if the items manufactured can take on fractional values.

C.1 Integer Programming

Variations of mathematical programming models differ primarily by making assumptions about the form of the objective function, feasible region or variable domains. As the assumptions change, the modeling power and tractability also change, usually in the opposite direction. The classic example is *integer programming* which differs from linear programming only in that some or all variables in its solutions are required to be integers:

Integer Programming model

$$
\begin{array}{lll}
\underset{\boldsymbol{x} \in R^n}{\text{minimize}} & c_1 x_1 \quad + \ldots + c_n x_n & \\
\text{subject to} & a_{11} x_1 \; + \ldots + a_{1n} x_n \leq b_1 & \\
 & \qquad \ldots & \\
 & a_{m1} x_1 + \ldots + a_{mn} x_n \leq b_m & \\
 & x_1, \ldots, x_n \geq 0. & \\
 & \text{One or more of } x_1, \ldots, x_n \text{ are integers.} &
\end{array}
$$

Integer programming is an NP-complete problem, illustrating that the increase in expressive power came at an enormous reduction in tractability. Integer programming is appropriate in roughly two types of situations. The first is when a decision variable domain contains mutually exclusive choices.

The *traveling salesman problem* is a well known NP-complete problem which can be cast as an integer programming problem where the variables indicating a path between two particular cities can be either zero or one. The second situation is when the decision variables refer to concrete physical items where it is nonsensical to have fractional values. The following is a simple example that illustrates this. (The example is adapted from Winston (1995) with different numbers to make the point illustrated in Figure C.1.)

Example: a simple problem for integer programming

> Denby Farms must decide how many acres of wheat and corn to plant this year. An acre of wheat yields \$295 of profit and corn \$885 per acre. An acre of wheat requires 13 hours of manual labor per week, and corn 14 hours. An acre of wheat yields 300 bushels and corn 220 bushels. They must supply their distributor with at least 1650 total bushels and have only 90 hours of labor available each week. We want to know many acres of each will maximize profits for Denby Farms.

The formulation of this problem as an integer programming problem is shown below in Figure C.1. If the items being produced were allowed to have fractional values, e.g. if pounds of sugar and gallons of vinegar were the items being manufactured, then linear programming could instead be applied. The integer restriction in integer programming solutions means that, in general, no guarantees can be made about the proximity of the optimal real-valued solution produced by the simplex algorithm, to the optimal integer-valued solution. The depiction of the solution space on the right in Figure C.1 illustrates this problem for the simple example. This problem exists regardless of whether the discrete decisions are integers or not.

$$
\begin{array}{lrll}
\text{maximize} & z = 295x_1 & + 885x_2 & \\
\boldsymbol{x} \in R^2 & & & \\
\text{subject to} & 13x_1 & + 14x_2 & \leq 90 \\
& -30x_1 & - 22x_2 & \leq -165 \\
& x_1, x_2 \geq 0 & &
\end{array}
$$

LP Answer: $x_1 = \frac{2075}{871}$, $x_2 = \frac{565}{134}$, $z = \$4434.31$

IP Answer: $x_1 = 5$, $x_2 = 1$, $z = \$2040.00$.

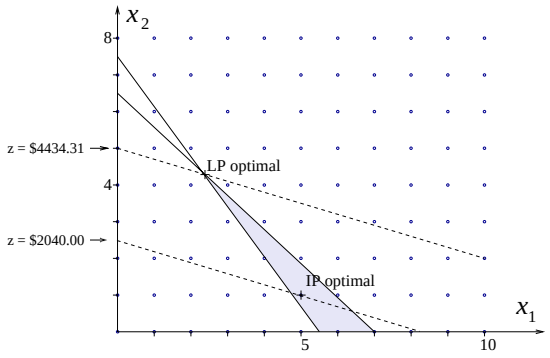


Figure C.1: The integer vs. fractional optima.

Since integer programming is an NP-complete problem, there will be no algorithms like simplex that allow a wide class of large sized problems to be readily solvable. But this doesn't mean that there aren't general methods available. One method available in problems where all the variables are restricted to have finite discrete domains, is an exhaustive search of the finite decision space,

where the decision space is formed from the Cartesian product of each variable's domain. Although an impractical approach to all but the smallest problems, it may be useful in conjunction with other methods.

A generally effective method for solving integer programming problems is the *branch-and-bound* method and its variations. The combinatorial explosion of choices in integer programming problems is structured as a decision tree. The algorithm branches through the tree while calculating bounds on potential solutions of sub-trees and eliminating whole sections without explicit consideration if the bounds eliminate hope for improving the solution.

The branch and bound method is also the general structure of the algorithms for solving IvP problems, as discussed in Chapter 5. The branch and bound method is more aptly described as a search method *architecture* since its true effectiveness relies on the bounding techniques, which are typically application specific. Certain techniques, such as the use of cutting planes, have general applicability, but the most effective branch-and-bound algorithms typically capitalize on the unique problem structure of the relevant application.

C.2 Nonlinear Programming

A nonlinear programming problem is one in which the objective function or one or more of the constraints is nonlinear. The general mathematical form is:

$$\begin{array}{lll} \underset{\boldsymbol{x} \in R^n}{\text{minimize}} & f_0(\boldsymbol{x}) & \text{Nonlinear Programming model} \\ \text{subject to} & f_i(\boldsymbol{x}) \leq 0, \qquad i = 1, \ldots, m & \end{array}$$

The structure of this form is not much more specific than the general optimization model. Consequently, not only are there no efficient methods like the simplex algorithm for solving nonlinear programs, there are no general methods at all, unless specific assumptions are made about the form of nonlinearity found. Usually, for particular applications, assumptions can in fact be made about the kinds of functions typically found. In control of autonomous vehicles however, these kinds of assumptions are very difficult due to the open-ended possibilities of acting in a real world setting.

C.3 Convex programming

Convexity plays a crucial role in the optimization of nonlinear functions. In a convex programming problem, the objective function and the feasible region are both convex. An important property of a convex objective function is that the value at a local extreme point is at least as good as the value at a global extreme, i.e. the function is *unimodal*. Unimodal functions are mentioned here because they generalize the class of convex functions, and if a class of applications cannot expect unimodal objective functions then it cannot expect convex objective functions. A function $(f : D \rightarrow R)$ is

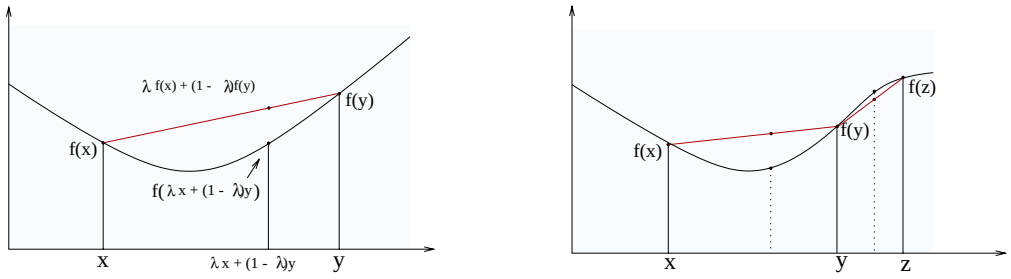
convex if the following inequality holds for any pair of points $\boldsymbol{x}, \boldsymbol{y} \in D$ and any real number $\lambda \in [0, 1]$:

$$f(\lambda \boldsymbol{x} + (1 - \lambda)\boldsymbol{y}) \leq \lambda f(\boldsymbol{x}) + (1 - \lambda) f(\boldsymbol{y})$$

and is *unimodal* if

$$f(\lambda \boldsymbol{x} + (1 - \lambda)\boldsymbol{y}) \leq \min(f(\boldsymbol{x}),\ f(\boldsymbol{y})).$$

The two functions in Figure C.2 show the difference between convex and unimodal functions. All convex functions are unimodal but not vice versa, and both have only one local and global minimum.



(a) A convex function

(b) A unimodal but nonconvex function

Figure C.2: Convex vs. Unimodal functions.

A function f is *concave* if $-f$ is convex. The convex programming model can be stated as:

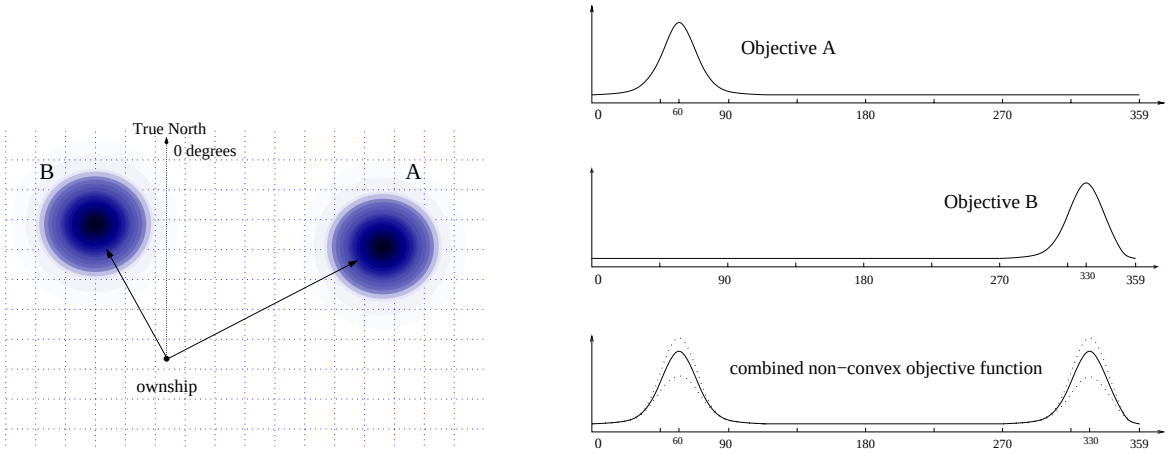
$$\begin{array}{lll} \text{minimize} \\ \boldsymbol{x} \in R^n & f_0(\boldsymbol{x}) & \text{Convex Programming} \\ & & \text{model} \\ \text{subject to} & f_i(\boldsymbol{x}) \leq 0, \qquad i = 1, \ldots, m & \end{array}$$

where $f_i : R^n \rightarrow R$ is convex, $i = 1, \ldots, m$.
$f_0 : D \rightarrow R$ is convex, $\boldsymbol{x} \in D$ iff $f_i(\boldsymbol{x}) \leq 0, \ \ i = 1, \ldots, m$.

If the aim is to *maximize* rather than minimize, the objective function must instead be concave, to have a local optimum that is global. Convex programming problems, regardless of the form otherwise of the objective function, are generally polynomial solvable. If further constraints on the objective function can be made, as in quadratic programming, then the prospects for tractible solution algorithms are considerably better.

Unimodal functions are not uncommon as output from a vehicle control behavior. Consider the objective of moving a vehicle to a desired location. Positions nearby the desired location are typically considered to be increasingly suboptimal as the position varies from the optimal point. In Figure

C.3(a), a situation is depicted from the AUV control problem where the controlled vehicle, ownship, must decide a next maneuver from its current position. The situation depicts two objectives, each resulting in two different optimal ocean positions for the AUV to move to. Each is represented by



(a) Two optimal lat-lon positions

(b) Combining objective functions

Figure C.3: Nonconvexity arises when adding two unimodal functions.

a unimodal objective function as shown in Figure C.3(a). However, when these objective functions are combined, by some prioritized additive combination, the unimodal property is lost, as shown in Figure C.3(b).

The multiple-objective aspect of typical autonomous vehicle control problems makes it very difficult to make assumptions about the form of the combined objective function, and to a lesser degree, the feasible region. If the methods from a particular single-objective mathematical programming model are to be used, the multiple objective functions must be combined in a way to make a single objective function. As the AUV example in Figure C.3 illustrates, this likely results in nonlinear, multimodal objective functions. The prospect for tractable solution methods are slim under these circumstances. The prospects are better if limitations can be made on the expected objective functions in a particular application. In a behavior-based implementation, using an action selection method with strong dependence on the objective function form is less than desirable for the reasons discussed in Section 2.2.

Appendix D

Executables and File Types

The information in this appendix is provided with two aims in mind. The first is to shed more light on the process used to produced the results given in Section 4.3 and Section 5.6, and the second is to facilitate the reproduction of these results, should anyone wish to later improve the algorithms and need a way of benchmarking against previous results.

In D.1, the executables and file types for creating the ring functions described in Section 4.2.1 and producing piecewise defined IvP functions is detailed. In D.2, the executables and file types for creating and solving IvP problems is detailed. Finally in D.3, the executable and file types for collecting data into LaTeXtables are described.

D.1 Creating Ring Functions and IvP Functions

There are two executables and two file types described here, which have the relationship shown in Figure D.1. The executable *makeRNGs* will create a group of ring functions and store them each in one file with the suffix ".rng". The executable *solveIPF* creates an IvP piecewise defined function

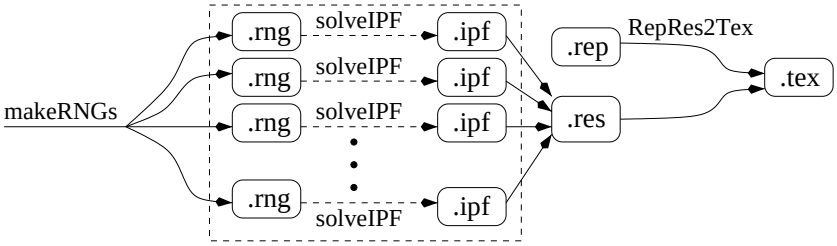


Figure D.1: Making ring functions and then IvP functions based on them.

based on the ring function in the input file. The output is stored in another file with the same name, but with the suffix ".ipf". The details of these executables and file types are discussed below in D.1.1 and D.1.2. By saving the .rng and .ipf files, later improvements to the algorithms in *solveIPF*

can be readily compared. Ultimately the output is a LaTeX table based on the results (in the .res file) and a report specification (in the .rep file), discussed in D.3.

D.1.1 Creating Ring Functions

Sets of ring functions are created using the executable `makeRNGs`. Each ring function is stored in a ring file, with the suffix ".rng". The usage for this function and the structure of .rng files is given in Table D.1. The first group of parameters to `makeRNGs`, `-amt` thru `-subdir` specify how many

Parameter	Description
-amt	Number of functions, i.e., files created
-index	Starting index (suffix) to file name
-prefix	Optional prefix to index suffix
-subdir	Directory to put .rng files
"ring parameters"	
-dim	Number of dimensions in each function
-dimsize(2)	Size of a variable domain
-alldim	Size of domain for all variables
-rings	Number of rings in each function
-range	Range for each ring component
-base	Base value for each ring component
-radiv	Divisor for setting radius by drop point
-radiv	Divisor for setting radius by drop point
-radRng(2)	Low/High range for random radius
-expRng(2)	Low/High range for random exponent
-rand	Re-seed the random number generator
-help	Display usage and default values

(a)

M	rings vars $D_0 \ldots D_{n-1}$
R	$x_0 \ldots x_{n-1}$ rad exp sign range base
	•
	•
R	$x_0 \ldots x_{n-1}$ rad exp sign range base

(b)

Table D.1: (a) Usage for the executable `makeRNGs`, (b) the .rng file format.

files/functions are to be made, and how they are to be named. Each generated file will have a file name beginning with the string provided to `-prefix`, followed by an index number starting with the number provided to the parameter `-index`. For example, the call `makeRNGs -amt 3 -index 4 -prefix ring` will create the three files: `ring_004.rng`, `ring_005.rng` and `ring_006.rng` The group of files is created in the same directory that the function is called from, unless a valid directory is supplied in the `-subdir` parameter.

The "ring parameters" The second group of parameters, `-dim` thru `-expRng`, specify characteristics that are used to generate each ring file. The parameters `-dim`, `-dimsize`, and `-alldim` determine the *universe* or domain of the function. The parameter `-dim` takes one argument, the number of dimensions, i.e., decision variables. The default is two variables. The parameter `-dimsize` takes two arguments: the variable index, and the integer size. The lower bound of the domain is always 0, and the upper bound's default is 850 unless a legal argument is provided. The parameter `-alldim` resets the default domain size of each decision variable. The parameter `-rings` determine how many rings in each function. The default is one ring. The `-base` and `-range` parameters specify

the range for each ring (within each ring function). The range for each ring function then is at most $[base, base + r(range)]$, where r is the number of rings.

There are four other factors randomly chosen for each ring generated: the center and radius of the ring, the exponent, and the sign. The center points are always randomly chosen to be somewhere in the universe given by the variables' domains, and the sign bit is 50/50 coin flip. The exponent is chosen randomly in the range given by the two arguments to the parameter `-expRng`. The default range is $[0, 20]$. Finally, the radius is chosen in one of two methods. If the parameter `-radiv` is provided, the radius is determined by picking a random point in the universe, calculating its distance to the center point, and dividing by a random number in the range $[1, r]$, where r is the number provided to the `-radiv` parameter. If `-radiv` is not provided, the radius is chosen randomly in the range given by the two arguments to the `-radRng` parameter. The default for that range is $[20, 100]$. The reason we keep two methods, is that sometimes we like to have control of the ring radius w.r.t. the universe size using the `-radiv` parameter, and sometimes we like to have explicit control of the ring size, using the `-radRng` parameter.

The "ring file" format Each ring file contains one line, prefixed with an M, with the number of rings, variables and variable domains specified. This is followed by r lines, each prefixed with an R, specifying the structure of each ring: first the $n-$vector indicating the center point, followed by the radius, exponent, sign, range, and base. The range and base are typically the same for each ring, but we chose to put this value on each line should this policy be later changed. An example ring file, with its corresponding ring function is given in Table D.2.

example.rng

M	3 2 800 800
R	42 698 47 6 1 200 -100
R	97 122 19 3 0 200 100
R	11 484 92 9 1 200 -100

$$f(x,y) = \left((1 - \frac{|\sqrt{(x-42)^2+(y-698)^2}-47|}{1600})^6 * 200\right) - 100 + \left((1 - \frac{|\sqrt{(x-97)^2+(y-122)^2}-19|}{1600})^3 * -200\right) + 100 + \left((1 - \frac{|\sqrt{(x-11)^2+(y-484)^2}-92|}{1600})^9 * 200\right) - 100$$

Table D.2: Example ring file with corresponding ring function.

D.1.2 Creating IvP Functions Based on Ring Functions

Interval programming (IvP) functions are created using the executable `solveIPF` taking a ring (.rng) file as input, and creating an .ipf file as output, while depositing various benchmark data in a results (.res) file. This relationship was depicted above in Figure D.1. The usage for this function and the structure of .ipf files is given in Table D.3. The first group of parameters to `solveIPF` specify the input (.rng) file specifying the ring function, the output (.ipf) file to contain the resulting IvP function, and a results (.res) file to contain data suitable for reporting benchmarks. If the input ring file is `ring_001.rng`, and the string "xyz" is provided to the `-res` parameter, the resulting IvP function will reside in the file `ring_001_xyz.ipf`, and the benchmark results will reside in the

Parameter	Description
-rng	Input file with ring function (.rng file)
-res	Output file with creation info (.res file)
-subdir	Directory to put .res files
"build parameters"	
-pcwise	Linear or scalar interior functions
-pieces	Max number of pieces in IvP function
-maxtime	Max time before termination
-avgerr(2)	Termination threshold for average error
-worerr(2)	Termination threshold for worst error
-selRand	Weight for "select random" policy
-selSize	Weight for "select biggest" policy
-selWrst	Weight for "select worst" policy
-queSize	Size of priority queue
-queSamp	Samples per iteration
"ring parameters" - see Table D.1	
-rand	Re-seed the random number generator
-help	Display usage and default values

(a)

M	rings vars $D_0 \ldots D_{n-1}$
R	$x_0 \ldots x_{n-1}$ rad exp sign range base
	•
	•
R	$x_0 \ldots x_{n-1}$ rad exp sign range base
F	count linearflag
B	$e_0^-\ e_0^+ \ldots e_{n-1}^-\ e_{n-1}^+\ m_0 \ldots m_{n-1}$ b
	•
	•
B	$e_0^-\ e_0^+ \ldots e_{n-1}^-\ e_{n-1}^+\ m_0 \ldots m_{n-1}$ b

(b)

Table D.3: (a) Usage for the executable `solveIPF`, (b) the .ipf file format.

file `xyz.res`. The output files are created in the same directory that the function is called from, unless a valid directory is supplied in the `-subdir` parameter.

The "build parameters" The second group of parameters, `-pcwise` thru `-queSamp`, specify characteristics used to build an IvP function from the given ring function. The parameter `-pcwise` determines whether the resulting IvP function will be piecewise constant, or piecewise linear. The default value is "constant" unless the argument "linear" is supplied. The integer value supplied to the `-pieces` parameter specifies a maximum number of pieces used in building the IvP function. It is typically the criteria used for halting the build algorithm.

There are three other halting criteria possible besides the limit on the number of pieces. If the `-maxtime` parameter is provided, in seconds, then the build algorithm will halt when this amount of time has been used up, or the max number of pieces has been reached, whichever first. The parameters `-avgerr` and `-worerr` specify the error threshold, below which the process will terminate. These each take two arguments. The first is the error threshold value, and the second is the number of sample points used in testing whether or not the threshold has been met. Typically the last two tests are performed after a fixed number of iterations through the select-split-refine loop described in Section 4.1.2. After the first of any of the four criteria is met, the process is terminated. If no criteria, i.e., stopping parameters, are provided, a default limit of 1000 pieces is in effect.

The parameters `-selRand` thru `-selWrst` determine which selection policy is used in the select-split-refine loop described in Section 4.1.2. By default each policy has an equal weight, 25, randomly switching between the three. Setting a value to zero turns off the policy completely, and proportionate weights reflect the proportion of times a particular policy is selected. The last pair of parameters relate to the priority queue maintained to keep track of the worst piece, selected for splitting and

refinement under the `-selWrst` policy. The parameter `-queSize` determines the size of the queue, and `-queSamp` determines how many points are sampled for insertion into the priority queue.

Final notes: The "ring parameters" listed in Table D.1 can also be provided to the `SolveIPF` executable to create a ring function "on the fly". The presence of any of ring parameters is ignored if a valid ring file is provided to the `-rng` parameter. And finally, if the `-res` parameter is not provided, the OF-Builder GUI, described in Section 7.1, is launched.

The "ipf file" format The first part of the .ipf file format shown in Table D.3 (b) contains the same ring function information present in the corresponding .rng file. This is included to ensure the tie to the underlying function from which the IvP function is derived, and because these few lines are relatively small compared to the typically thousands of lines describing the IvP function. The first line in the second part, with the F prefix, contains, first the count of pieces in the piecewise function, indicating the number of lines to follow, and second, a flag indicating whether the interior function of each piece is a linear or scalar function. Each line, prefixed with B in the .ipf file, indicates a single piece in the piecewise defined IvP function. The first n pairs, one for each variable, indicate the interval over which it is defined. The last n floating point values indicate the slopes and intercept value of the corresponding interior linear function. A single scalar value follows the intervals if the interiors are scalar functions.

D.2 Creating and Solving IvP Problems

There are two executables and one file type described here, which have the relationship shown in Figure D.2. The executable `makeIPPs` will create a group of IvP problems and store them each in a

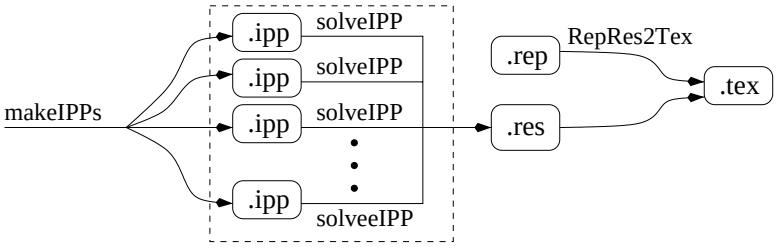


Figure D.2: Making IvP problems, solving them, and collecting the results.

unique file with the suffix ".ipp". The executable `solveIPP` solves an IvP problem based on an input (.ipp) file, and stores the results in results (.res) file. The details of these two executables and the .ipp file type are discussed below in D.2.1 and D.2.2. Ultimately the output is a LaTeXtable based on the results (in the .res file) and a report specification (in the .rep file), discussed in D.3

D.2.1 Creating IvP Problems

Sets of IvP problems are created using the executable `makeIPPs`. Each problem instance is stored in an IvP problem (.ipp) file. The usage for this function and the structure of .ipp files is given in Table D.4. Each set of parameters to `makeIPPs` is similar to a set described above for either `makeRNGs` or

Parameter	Description
-amt	Number of problems, i.e., files created
-index	Starting index (suffix) to file name
-prefix	Optional prefix to index suffix
-subdir	Directory to put .ipp files
-ofs	Number of functions in each problem
"ring parameters" - see Table D.1	
"build parameters" - see Table D.3	
-rand	Re-seed the random number generator
-help	Display usage and default values

(a)

P	#-of-OFs
F	count vars pwt linearflag $D_0 \ldots D_{n-1}$
B	$e_0^- \; e_0^+ \ldots e_{n-1}^- \; e_{n-1}^+ \; m_0 \ldots m_{n-1}$ b
	•
	•
B	$e_0^- \; e_0^+ \ldots e_{n-1}^- \; e_{n-1}^+ \; m_0 \ldots m_{n-1}$ b
	•
	•
F	count vars pwt linearflag $D_0 \ldots D_{n-1}$
B	$e_0^- \; e_0^+ \ldots e_{n-1}^- \; e_{n-1}^+ \; m_0 \ldots m_{n-1}$ b
	•
	•
B	$e_0^- \; e_0^+ \ldots e_{n-1}^- \; e_{n-1}^+ \; m_0 \ldots m_{n-1}$ b

(b)

Table D.4: (a) Usage for the executable `makeIPPs`, (b) the .ipp file format.

`solveIPF`. This is because the executable `makeIPPs` is composed of these two processes.

The first group of parameters to `makeIPPs`, `-amt` thru `-subdir` specify how many files/problems are to be made, and how they are to be named. Each generated file will have a file name beginning with the string provided to `-prefix`, followed by an index number starting with the number provided to the parameter `-index`. For example, the call `makeIPPs -amt 3 -index 4 -prefix prob` will create the three files: `prob_004.ipp`, `prob_005.ipp` and `prob_006.ipp` The group of files is created in the same directory that the function is called from, unless a valid directory is supplied in the `-subdir` parameter.

The "ring parameters" describe the attributes for the underlying ring functions (see Table D.1), and the "build parameters" specify how IvP functions are to be made from each ring function (see Table D.3). The `-ofs` parameter is the only new one unique to this function and indicates how many objective functions are to be present in each problem instance.

The ".ipp file" format An .ipf file contains three different types of lines. The first line, prefixed with a P, has one piece of information: the number of objective functions in the problem. This line is followed by several groups of lines, one for each objective function. The first line in each group, prefixed with an F, indicates the number of pieces in its objective function, the number of variables, the priority weight of the function, a flag indicating the type of interior function, and finally a domain size for each of the n variables. The lines prefixed with B, one for each piece, have the same format as in .ipf files described in Table D.3. The first n pairs, one for each variable, indicates the interval over which it is defined. The last n floating point values indicate the slopes and intercept value of the corresponding interior linear function. A single scalar value follows the

intervals if the interiors are scalar functions.

D.2.2 Solving IvP Problems

Interval programming problems are solved using the `solveIPP` executable given in Table D.5. It solves the problem given in the .ipp file using the algorithm given to the `-alg` parameter. There

Parameter	Description
-ipp	Input file with IvP problem (.ipp file)
-res	Output file with solution info (.res file)
-subdir	Directory to put .res files
-alg	Algorithm used to solve IPP problem
-ofs	Number of functions in each problem
"ring parameters" - see Table D.1	
"build parameters" - see Table D.3	
-rand	Re-seed the random number generator
-help	Display usage and default values

Table D.5: Usage for the executable: solveIPP

are three algorithms available: "bb" for branch-and-bound, "bf" for brute force, and "mc" for max-clique. The max-clique algorithm was discussed in Section 5.5. An output file, specified by the `-res` parameter, will be written to with solution and benchmark information. Like the executable `solveIPF`, if the `-res` parameter is not provided, the IvP -Solver GUI, discussed in Section 7.1 is launched.

In collecting data for benchmarks, `solveIPP` is typically given a previously built .ipp file. More often, the solveIPP is executed on a problem created "on the fly" at execution time. Thus the "ring parameters" (Table D.1), and "build parameters" (Table D.3) and a value for `-ofs` can be used to create such a function. These parameters will be ignored if a valid .ipp file is provided to the parameter `-ipp`.

D.3 Collecting and Reporting Results

Usage for the executable: RepRes2Tex

Parameter	Description
-rep	A report style file (.rep file)
-res	A results file (.res file)
-tex	Output LaTeX file (.tex) file

Appendix E

Notes on Empirical Results for IvP Function Creation

The purpose of this appendix is to provide an expanded set of results for the three experiments reported in Sections 4.3.1 thru 4.3.3, which correspond here to Sections E.2 thru E.4. In addition, a sufficient level of detail is provided concerning the design of the experiment to facilitate reproduction. The issue of reproducing results was addressed in Section 4.2.3

E.1 Conditions Common to All Experiments

All code was compiled using g++ in a Linux environment with the standard g++ optimizer (-O) flags set. All experiments were run on a Pentium III platform with a CPU speed of 1004MHz, 2.06 gigabytes of RAM, and a cache size of 250 kilobytes. All reports of time are in seconds, and measure *cpu_time* (not *wall_time*, *user_cpu_time*, or *system_cpu_time*).

E.2 Experiment 1: Accuracy vs. Resources vs. Time

This section contains details on the experiment reported in Section 4.3.1. For more on the meaning of the parameters to the `makeRNGs` executable, or the `solveIPF` executable, see Appendix A. For more on the method of reporting error, see Section 4.2.2. For more on ring functions, see Section 4.2.1. The shaded cells in the tables below indicate data plotted in Section 4.3.1. The non-shaded cells indicate expanded results not previously plotted.

E.2.1 Design of the experiment

In total, 25 different ring functions were created for this experiment. Each were created with the same parameters, allowing only the position, radius, and slope of each ring component to vary radomly.

Ring Functions for This Experiment

Location	/DATA_DIR/Files-rngs/GroupBeta/EXP_003
Ring files:	beta_001.rng ... beta_050.rng
makeRNGs	-prefix beta amt 25 -index 1 -rings 4 -dim 4 -expRng 0 20 -radiv 50 -range 200 -base N100 -alldim 850

For each of the 25 ring functions, we created non-uniform IvP functions using the algorithm described in Section 4.1.2. IvP functions with uniform pieces are not considered here. We created 5 distinct groups of results by invoking `solveIPF` with 5 different settings to the parameter `-pieces` ranging from 1000 to 20000, as indicate below in Table E.1. Within each of these 5 groups we varied the number of sample points by varying the value given to the `-queSamp` parameter upon invoking `solveIPF`. For the 1000-piece group, the following values for `-queSamp` were used: (1, 3, 6, 8, 10, 14, 20, 28, 35, 50, 60, 75, 90, 120, 150, 180, 210, 240, 270, 300, 350, 400, 450, 500, 600, 700). Only the first ten of these values were used in the 20000-piece group. The value given to the parameter `-queSamp` determines the number of sample points in each pass of the loop described in Section 4.1.2. This approach used a fixed-length priority queue, which we set to be 128 for all runs in this experiment. In total, the results of 2425 different invocations of `solveIPF` are report here.

E.2.2 Expanded results of the experiment

In Table E.1 the five shaded columns contain the data plotted in Figure 4.7 in Section 4.3.1. We also report the average error (`avgErr`) for comparison. See Section 4.2.2 for the difference between the two. Note that the average error does not improve significantly with the increase in time, i.e., number of sample points.

	pieces=1000		pieces=2000		pieces=5000		pieces=10000		pieces=20000	
createTm	avgErr	rptErr	avgErr	rptErr	avgErr	rptErr	avgErr	rptErr	avgErr	rptErr
4.00	0.46	2.15	0.33	2.48	-	-	-	-	-	-
8.00	0.48	1.92	0.33	1.69	-	-	-	-	-	-
12.00	0.49	1.90	0.33	1.48	-	-	-	-	-	-
16.00	0.49	1.91	0.34	1.36	0.20	1.72	-	-	-	-
20.00	0.50	1.89	0.34	1.34	0.19	1.18	-	-	-	-
24.00	0.51	1.89	0.34	1.35	0.20	1.07	-	-	-	-
28.00	-	-	0.34	1.36	0.19	0.91	-	-	-	-
32.00	0.52	1.93	0.35	1.29	0.21	0.98	0.15	1.64	-	-
36.00	0.53	1.96	-	-	0.20	0.96	0.14	1.11	-	-
40.00	-	-	0.35	1.31	0.19	0.91	0.14	0.83	-	-
44.00	-	-	0.35	1.31	0.21	0.87	0.14	0.85	-	-
48.00	-	-	0.35	1.31	0.19	1.09	0.14	0.81	-	-
52.00	-	-	0.35	1.29	0.21	0.89	0.14	0.70	-	-
60.00	-	-	-	-	0.21	0.83	0.15	0.70	-	-
64.00	-	-	-	-	-	-	-	-	0.12	1.40
68.00	-	-	-	-	0.21	0.83	0.15	0.69	0.11	1.26
72.00	-	-	-	-	0.23	0.79	-	-	0.10	0.83
76.00	-	-	-	-	0.20	0.95	0.15	0.67	0.10	0.73
80.00	-	-	-	-	0.22	0.80	-	-	0.10	0.69
84.00	-	-	-	-	0.20	0.84	0.15	0.63	0.10	0.69
88.00	-	-	-	-	0.21	0.79	-	-	-	-
92.00	-	-	-	-	0.22	0.90	-	-	0.10	0.60
96.00	-	-	-	-	0.21	0.80	-	-	-	-
100.00	-	-	-	-	-	-	0.14	0.72	-	-
104.00	-	-	-	-	0.21	0.81	0.15	0.61	0.10	0.61
108.00	-	-	-	-	0.23	0.77	-	-	-	-
112.00	-	-	-	-	-	-	-	-	0.10	0.50
116.00	-	-	-	-	-	-	0.13	0.70	0.12	0.55
120.00	-	-	-	-	-	-	0.15	0.61	-	-
132.00	-	-	-	-	-	-	-	-	0.09	0.60
136.00	-	-	-	-	-	-	0.15	0.64	0.11	0.53
140.00	-	-	-	-	-	-	0.16	0.55	-	-

Table E.1: Combined error and average error vs. time (in seconds).

E.3 Experiment 2: Linear vs. Constant w.r.t. Dimension Size

This section contains details on the experiment reported in Section 4.3.2. For more on the meaning of the parameters to the `makeRNGs` executable, or the `solveIPF` executable, see Appendix A. For more on the method of reporting error, see Section 4.2.2. For more on ring functions, see Section 4.2.1. The shaded cells in the tables below indicate data plotted in Section 4.3.2. The non-shaded cells indicate expanded results not previously plotted.

E.3.1 Design of the experiment

This experiment has two parts, but both are based on the same sets ring functions. The first part of the experiment compares the accuracy of piecewise constant vs. piecewise linear functions as the

number of dimensions grow. We created 9 groups of 10 functions, ranging from 2D to 20D:

Ring Functions for This Experiment

Location	/DATA_DIR/Files-rngs/GroupGamma/EXP_001
Ring files:	gamma2D_001.rng ... gamma2D_010.rng gamma3D_001.rng ... gamma3D_010.rng gamma4D_001.rng ... gamma4D_010.rng gamma5D_001.rng ... gamma5D_010.rng gamma8D_001.rng ... gamma8D_010.rng gamma10D_001.rng ... gamma10D_010.rng gamma12D_001.rng ... gamma12D_010.rng gamma15D_001.rng ... gamma15D_010.rng gamma20D_001.rng ... gamma20D_010.rng
(all groups)	makeRNGs -amt 10 -index 1 -rings 4 -range 200 -base N100 -radiv 50 -expRng 0 20 -rand -alldim 850
Group 2D:	makeRNGs -prefix gamma2D -dim 2
Group 3D:	makeRNGs -prefix gamma3D -dim 3
Group 4D:	makeRNGs -prefix gamma4D -dim 4
Group 5D:	makeRNGs -prefix gamma5D -dim 5
Group 8D:	makeRNGs -prefix gamma8D -dim 8
Group 10D:	makeRNGs -prefix gamma10D -dim 10
Group 12D:	makeRNGs -prefix gamma12D -dim 12
Group 15D:	makeRNGs -prefix gamma15D -dim 15
Group 20D:	makeRNGs -prefix gamma20D -dim 20

In the first part of the experiment, for each of these 90 functions, we invoked `solveIPF` with 6 different parameter settings: three as piecewise constant, and three as piecewise linear. Within each three, one was created with 3000 pieces, one with 8000, and one with 15000 pieces. The results of a total of 540 runs of `solveIPF` are reported here. In the second part of the experiment, we focussed on the 8D ring functions, and varied the number of pieces from 100 thru 25000, for both piecewise constant and piecewise linear functions.

E.3.2 Expanded results of the experiment

In Table E.2 the six shaded columns contain the data plotted in Figure 4.8 in Section 4.3.2.

EXP-GO1.rep1

	pieces=3000		pieces=8000		pieces=15000	
	pcwise=C	pcwise=L	pcwise=C	pcwise=L	pcwise=C	pcwise=L
dim	reportErr					
2.00	1.17	0.29	0.74	0.18	0.54	0.13
3.00	3.29	1.00	2.55	0.56	2.12	0.33
4.00	5.34	1.80	4.29	1.47	3.91	1.19
5.00	7.69	3.35	6.57	2.35	5.41	1.74
8.00	11.50	6.03	10.50	5.29	9.46	4.21
10.00	13.98	7.55	12.82	6.48	12.53	6.16
12.00	14.88	7.68	13.53	7.11	12.87	6.19
15.00	19.11	9.78	17.19	8.87	15.96	8.05
20.00	29.89	11.15	26.95	10.48	25.23	9.65

Table E.2: Combined error vs. number of dimensions.

In Table E.3 the *average error* is reported for the same set of experiments.

EXP-GO1.rep1

	pieces=3000		pieces=8000		pieces=15000	
	pcwise=C	pcwise=L	pcwise=C	pcwise=L	pcwise=C	pcwise=L
dim	avgErr					
2.00	0.40	0.01	0.24	0.00	0.18	0.00
3.00	1.05	0.09	0.77	0.05	0.62	0.03
4.00	1.72	0.29	1.34	0.18	1.16	0.13
5.00	2.21	0.50	1.88	0.36	1.63	0.27
8.00	3.02	1.08	2.66	0.88	2.42	0.76
10.00	2.84	1.00	2.64	0.90	2.51	0.83
12.00	3.32	1.21	2.99	1.00	2.85	0.94
15.00	4.00	1.47	3.46	1.25	3.14	1.10
20.00	6.63	1.66	5.95	1.53	5.56	1.45

Table E.3: Average error vs. number of dimensions.

In Table E.4 the *creation time* is reported for the same set of experiments.

EXP-GO1.rep1

	pieces=3000		pieces=8000		pieces=15000	
	pcwise=C	pcwise=L	pcwise=C	pcwise=L	pcwise=C	pcwise=L
dim	createTm					
2.00	0.80	3.05	2.24	8.25	4.34	15.61
3.00	1.05	6.14	2.90	16.59	5.34	30.96
4.00	1.30	9.74	3.51	26.02	7.48	49.61
5.00	1.30	16.41	3.54	43.83	9.44	84.89
8.00	2.60	40.42	8.08	108.86	17.39	206.27
10.00	2.15	56.34	5.81	150.35	25.82	296.60
12.00	2.50	76.08	6.66	202.94	12.55	380.56
15.00	3.13	111.12	8.15	296.36	15.21	555.47
20.00	9.19	188.51	16.28	495.69	25.72	924.27

Table E.4: Creation time vs. number of dimensions.

In Table E.5 the creation time (`createTm`) and combined error (`repErr`) are reported in relation to a growing piece count. The two shaded columns contain the data plotted in Figure 4.9 in Section 4.3.2.

	dim=8			
	pcwise=C	pcwise=L	pcwise=C	pcwise=L
pieces	reportErr		createTm	
100.00	17.43	**10.25**	0.07	**1.33**
250.00	14.69	8.68	0.16	3.33
500.00	13.89	8.03	0.31	6.66
750.00	13.08	7.11	0.46	9.97
1000.00	13.08	7.14	0.62	13.30
2000.00	11.84	6.64	1.22	26.60
3000.00	11.61	6.21	2.50	40.69
5000.00	**11.31**	5.62	**4.43**	68.17
7000.00	10.48	5.58	6.54	95.86
9000.00	9.87	4.83	8.81	123.73
12000.00	9.48	4.53	12.40	165.79
15000.00	9.66	4.50	16.30	208.12
20000.00	9.03	4.10	23.37	279.37
25000.00	8.59	3.76	30.87	351.13

Table E.5: Creation time and `repErr` vs. piece count.

E.4 Experiment 3: Uniform vs. Non-uniform Pieces

This section contains details on the experiment reported in Section 4.3.3. For more on the meaning of the parameters to the `makeRNGs` executable, or the `solveIPF` executable, see Appendix A. For more on the method of reporting error, see Section 4.2.2. For more on ring functions, see Section 4.2.1. The shaded cells in the tables below indicate data plotted in Section 4.3.3. The non-shaded cells indicate expanded results not previously plotted.

E.4.1 Design of the experiment

In total, 120 different ring functions were created for this experiment. Note the parameters next to "(all groups)" below. In each ring function, there are 10 rings, and the `-radRng` parameter is set to zero. This will result in at most 10 separate local optima. We did not test in this experiment how the results would vary as the number of optima vary.

Ring Functions for This Experiment

Location	/DATA_DIR/Files-rngs/GroupEpsilon/EXP_001
Ring files:	epsilon2DA_001.rng ... epsilon2DA_020.rng epsilon2DB_001.rng ... epsilon2DB_020.rng epsilon2DC_001.rng ... epsilon2DC_020.rng epsilon3DA_001.rng ... epsilon3DA_020.rng epsilon3DB_001.rng ... epsilon3DB_020.rng epsilon3DC_001.rng ... epsilon3DC_020.rng
(all groups)	makeRNGs -amt 20 -index 1 -rings 10 -range 200 -rand -radRng 0 0 -base N10 -alldim 200
Group 2DA:	makeRNGs -prefix epsilon2DA -dim 2 -expRng 25 25
Group 2DB:	makeRNGs -prefix epsilon2DB -dim 2 -expRng 75 75
Group 2DC:	makeRNGs -prefix epsilon2DC -dim 2 -expRng 150 150
Group 3DA:	makeRNGs -prefix epsilon3DA -dim 3 -expRng 25 25
Group 3DB:	makeRNGs -prefix epsilon3DB -dim 3 -expRng 50 50
Group 3DC:	makeRNGs -prefix epsilon3DC -dim 3 -expRng 150 150

Within these six groups, the experiment varied over the number of pieces used, the uniformity or non-uniformity, and the interior function (linear vs. constant). In total there were 2520 test runs (6 groups of 20 ring functions, and 21 variations on `solveIPF` for each function). The choice for piece count, in Tables E.6 thru E.9, may seem a bit arbitrary at first glance. They were chosen for the following reason. Each decision variable had a domain of size 200, i.e., $[0, 199]$. In the case were uniform piecewise functions were built, the `solveIPF` executable was given a value for the `-unif` parameter. Seven different values were given for this parameter: (20, 15, 12, 10, 8, 6, 5). This resulted in the seven different piece counts, 100 thru 1600 in 2D, and 1000 thru 64000 in 3D. For example, $\lceil 200/15 \rceil = 14$, and $14^2 = 196$, and $14^3 = 2744$, would explain the piece counts in the second lines of Tables E.6 and E.7. When creating the non-uniform functions, `solveIPF` is passed the corresponding piece count to the `-pieces` parameter.

E.4.2 Expanded results of the experiment

As mentioned in Section 4.3.3, only one of the six groups were plotted in Figure 4.10. The other combined error is reported for the other five groups below in Table E.6 for 3 dimensions and Table E.7 for 2 dimensions.

Reporting Combined Error

	dim=3								
	exp=25			exp=75			exp=150		
	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin
pieces	reportErr								
1000.00	6.59	3.77	3.16	9.57	8.87	6.48	12.90	13.25	10.13
2744.00	4.89	2.74	1.77	8.51	7.26	4.43	12.35	11.57	8.24
4913.00	4.25	2.07	1.37	7.65	5.74	3.81	10.63	10.20	6.04
8000.00	3.47	1.66	1.24	6.41	4.96	2.84	10.13	7.92	5.33
15625.00	2.99	1.23	0.93	5.98	3.69	2.41	10.59	6.74	4.73
39304.00	2.28	0.86	0.59	5.00	2.80	1.62	8.72	4.38	3.51
64000.00	1.85	0.67	0.45	3.83	1.85	1.67	8.16	3.84	3.02

Table E.6: Combined error vs. piece count - for 3 dimensions.

In Table E.7 the same experiment is run on similar ring functions in two dimensions.

	dim=2								
	exp=25			exp=75			exp=150		
	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin
pieces	reportErr								
100.00	9.89	5.40	4.62	12.19	10.21	8.38	12.88	12.63	11.57
196.00	7.86	3.79	2.86	11.21	8.45	5.68	12.61	11.38	9.18
289.00	6.77	3.03	2.38	10.25	7.24	4.69	12.11	10.40	7.50
400.00	5.67	2.42	1.76	8.96	6.30	4.43	11.36	9.55	6.57
625.00	4.79	1.91	1.38	8.20	5.15	3.67	10.89	8.40	6.11
1156.00	3.63	1.37	0.97	6.74	3.77	2.73	9.82	6.56	5.15
1600.00	2.88	1.20	0.83	6.22	3.37	2.47	9.23	6.09	4.30

Table E.7: Combined error vs. piece count - for 2 dimensions.

Reporting average error

In Table E.8 the results are shown for the same experiments as reported in Table E.6 (the three sets of 3D functions) except that we are reporting average error (`avgErr`) instead of the combined error (`repErr`). See Section 4.2.2 for more on the difference between these two error measurements.

	dim=3								
	exp=25			exp=75			exp=150		
	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin
pieces	avgErr								
1000.00	0.60	0.17	0.17	0.16	0.10	0.06	0.05	0.05	0.02
2744.00	0.45	0.09	0.09	0.12	0.06	0.03	0.04	0.03	0.01
4913.00	0.36	0.06	0.06	0.10	0.04	0.02	0.03	0.02	0.01
8000.00	0.30	0.04	0.04	0.08	0.03	0.02	0.03	0.02	0.01
15625.00	0.25	0.03	0.03	0.07	0.02	0.01	0.02	0.01	0.00
39304.00	0.19	0.01	0.01	0.05	0.01	0.01	0.02	0.01	0.00
64000.00	0.15	0.01	0.01	0.04	0.01	0.00	0.01	0.00	0.00

Table E.8: Average error vs. piece count - for 3 dimensions.

Reporting creation time

In Table E.9 the results are shown for the same experiments as reported in Table E.6 (the three sets of 3D functions) except that we are reporting create time (`createTm`) instead of the combined error (`repErr`).

	dim=3								
	exp=25			exp=75			exp=150		
	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin	Unif-Const	Unif-Lin	NonU-Lin
pieces	createTm								
1000.00	0.09	2.14	4.80	0.09	2.14	4.87	0.09	2.15	4.88
2744.00	0.23	5.90	13.21	0.24	5.91	13.39	0.24	5.91	13.38
4913.00	0.42	10.60	23.48	0.42	10.54	23.83	0.42	10.55	23.84
8000.00	0.69	17.20	38.66	0.69	17.15	**39.14**	0.69	17.16	39.20
15625.00	1.35	33.40	75.88	1.35	**33.53**	76.82	1.35	33.54	76.85
39304.00	3.39	83.06	191.73	3.38	84.34	193.54	3.38	84.40	193.34
64000.00	5.51	134.91	313.69	5.50	137.21	316.38	5.51	137.34	314.64

Table E.9: Creation time vs. piece count - for 3 dimensions.

Appendix F

Notes on Empirical Results for IvP Problem Solutions

The purpose of this appendix is to provide an expanded set of results for the five experiments reported in Sections 5.6.1 thru 5.6.4, which correspond here to Sections F.2 thru F.5. In addition, a sufficient level of detail is provided concerning the design of the experiment to facilitate reproduction. The issue of reproducing results was addressed in Section 4.2.3

F.1 Conditions Common to All Experiments

All code was compiled using g++ in a Linux environment with the standard g++ optimizer (-O) flags set. All experiments were run on a Pentium III platform with a CPU speed of 1004MHz, 2.06 gigabytes of RAM, and a cache size of 250 kilobytes. All reports of time are in seconds, and measure *cpu_time* (not *wall_time*, *user_cpu_time*, or *system_cpu_time*).

F.2 Experiment 1: Plane Sweep Search vs. IvP Methods

F.2.1 Design of the Experiment

In this experiment, we created groups of 10 problems with the piece count ranging from 1,000 to 50,000 pieces, with 5 and 10 objective functions. The results are shown in Figure 5.18 and Table F.1.

F.2.2 Expanded Results of the Experiment

	Plane-sweep		Branch-Bound	
	ofs=5	ofs=10	ofs=5	ofs=10
pieces	time (seconds)			
1000.00	0.07	0.17	0.01	0.03
3000.00	0.24	0.57	0.02	0.05
5000.00	0.42	1.01	0.03	0.08
7000.00	0.60	1.50	0.05	0.11
10000.00	0.88	2.27	0.07	0.14
15000.00	1.39	3.83	0.09	0.21
20000.00	1.92	5.57	0.13	0.27
30000.00	3.08	9.35	0.17	0.41
40000.00	4.34	13.42	0.23	0.56
50000.00	5.72	17.80	0.30	0.70

Table F.1: Expanded results for the plane-sweep vs. branch-and-bound experiment.

F.3 Experiment 2: Solution Time vs. Number of Dimensions

F.3.1 Design of the Experiment

For this experiment, groups of 10 random functions were created, each with 10,000 pieces, and 5 objective functions, and each based on 5 ring functions. The results are shown in Figure 5.19, and in Table F.2.

F.3.2 Expanded Results of the Experiment

	pcwise=L					pcwise=C				
dim	time	totalGels	gelsze	ginsert	gquery	time	totalGels	gelsze	ginsert	gquery
2.00	0.17	4356.00	13.00	0.11	0.03	0.14	4356.00	13.00	0.08	0.02
3.00	0.19	5832.00	50.00	0.12	0.02	0.15	5832.00	50.00	0.10	0.02
4.00	0.23	4096.00	110.00	0.14	0.03	0.17	4096.00	110.00	0.10	0.03
5.00	0.23	3125.00	180.00	0.15	0.03	0.18	3125.00	180.00	0.12	0.03
6.00	0.31	4096.00	215.00	0.16	0.05	0.20	4096.00	215.00	0.13	0.03
7.00	0.31	2187.00	300.00	0.18	0.06	0.21	2187.00	300.00	0.15	0.04
8.00	0.39	6561.00	300.00	0.19	0.07	0.22	6561.00	300.00	0.15	0.03
9.00	0.88	512.00	426.00	0.20	0.47	0.23	512.00	426.00	0.16	0.05
10.00	0.75	1024.00	426.00	0.22	0.28	0.26	1024.00	426.00	0.19	0.04

Table F.2: Expanded results for the solution time vs. number of dimensions experiment.

The anomaly shown above, and previously in Figure 5.19, for the 9D piecewise linear case can be explained with the following explanation. In each problem, an initial number of pieces were dedicated in each IvP function in a uniform manner, with the remaining pieces distributed non-uniformly. See Section 5.2.5 for the motivation behind this. Furthermore, the grids used in the IvP solution algorithms were aligned with the size of the pieces used in the initial uniform functions.

The number of pieces chosen for the initial uniform function were based on the dth root of 10,000 where d is the number of dimensions, and 10,000 pieces is the target number of total pieces. In 9D, the largest this number can be is 512, since $2^9 = 512$ and $3^9 = 19683$, which is greater than 10,000. Since 512 is a relatively small portion of 10,000, the effectiveness of semi-uniform functions is minimized, thus resulting in a higher solution time than for its 10D counterpart which uses 1024 pieces in its semi-uniform functions.

F.4 Experiment 3: Solution Time vs. Number of IvP pieces

F.4.1 Design of the Experiment

In this experiment, groups of 10 random functions were created, each in 4 dimensions, and with 5 objective functions. The results are shown in Figure 5.20, and in Table F.3.

F.4.2 Expanded Results of the Experiment

	pcwise=L					pcwise=C				
pieces	time	totalGels	gelsze	ginsert	gquery	time	totalGels	gelsze	ginsert	gquery
1000.00	0.02	625.00	171.00	0.01	0.00	0.02	625.00	171.00	0.01	0.00
3000.00	0.07	1296.00	142.00	0.04	0.01	0.05	1296.00	142.00	0.03	0.01
5000.00	0.11	2401.00	122.00	0.06	0.02	0.08	2401.00	122.00	0.05	0.01
10000.00	0.23	4096.00	107.00	0.14	0.03	0.17	4096.00	107.00	0.12	0.02
15000.00	0.32	6561.00	95.00	0.20	0.05	0.26	6561.00	95.00	0.18	0.04
20000.00	0.44	10000.00	86.00	0.28	0.06	0.34	10000.00	86.00	0.23	0.04
30000.00	0.65	14641.00	78.00	0.42	0.09	0.52	14641.00	78.00	0.34	0.07
40000.00	0.88	20736.00	71.00	0.59	0.11	0.69	20736.00	71.00	0.45	0.09
50000.00	1.08	28561.00	66.00	0.73	0.13	0.86	28561.00	66.00	0.58	0.12
75000.00	1.64	38416.00	61.00	1.11	0.20	1.30	38416.00	61.00	0.87	0.16
100000.00	2.18	50625.00	57.00	1.47	0.27	1.73	50625.00	57.00	1.19	0.23

Table F.3: Expanded results for the solution time vs. number of IvP pieces experiment.

F.5 Experiment 4: Solution Time vs. Number of Objective Functions

F.5.1 Design of the Experiment

In this experiment, groups of 10 random functions were created, each in 4 dimensions, and with 5,000 pieces. The results are shown in Figure 5.21.

F.5.2 Expanded Results of the Experiment

	pcwise=L					pcwise=C				
ofs	time	totalGels	gelsze	ginsert	gquery	time	totalGels	gelsze	ginsert	gquery
2.00	0.10	4096.00	107.00	0.05	0.02	0.08	4096.00	107.00	0.04	0.02
3.00	0.15	4096.00	107.00	0.08	0.03	0.11	4096.00	107.00	0.07	0.03
5.00	0.23	4096.00	107.00	0.14	0.03	0.17	4096.00	107.00	0.11	0.02
7.00	0.32	4096.00	107.00	0.20	0.04	0.23	4096.00	107.00	0.16	0.03
10.00	0.63	4096.00	107.00	0.29	0.09	0.31	4096.00	107.00	0.22	0.02
15.00	0.84	4096.00	107.00	0.44	0.12	0.46	4096.00	107.00	0.34	0.03
20.00	1.73	4096.00	107.00	0.60	0.24	0.60	4096.00	107.00	0.46	0.02
25.00	2.82	4096.00	107.00	0.67	0.41	0.74	4096.00	107.00	0.60	0.03
30.00	4.33	4096.00	107.00	0.84	0.58	0.90	4096.00	107.00	0.70	0.03
40.00	6.73	4096.00	107.00	1.18	0.81	1.19	4096.00	107.00	0.93	0.05
50.00	12.35	4096.00	107.00	1.37	1.19	1.50	4096.00	107.00	1.19	0.06

Table F.4: Expanded results for the solution time vs. number of objective functions experiment.

Bibliography

Philip E. Agre and D. Chapman. What Are Plans For? *Robotics and Autonomous Systems*, 6:17–34, 1990.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

Ronald A. Arkin. Motor Schema-Based Mobile Robot Navigation. *International Journal of Robotics Research*, 8(4):92–112, 1989a.

Ronald A. Arkin. Behavior-Based Robot Navigation for Extended Domains. *Adaptive Behavior*, 1 (2):201–225, 1992.

Ronald C. Arkin. Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 264–271, Raleigh, NC, 1987.

Ronald C. Arkin. Towards the Unification of Navigational Planning and Reactive Control. In *AAAI Spring Symposium on Robot Navigation*, 1989b.

Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.

Michael R. Benjamin. Underwater Vehicle Control: Minimum Requirements for a Robust Decision Space. In *Command and Control Research and Technology Symposium*, Naval Postgraduate School, Monterey, CA, June 2000a.

Michael R. Benjamin. Virtues and Limitations of Voting Based Action Selection. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents 2000)*, Barcelona, Spain, 2000b.

Michael R. Benjamin, Thomas Viana, Karen Corbett, and Ann Silva. Satisfying Multiple Rated-Constraints in a Knowledge Based Decision Aid. In *Proceedings IEEE Conference on Artificial Intelligence Applications*, Orlando, FL, 1993.

Andrew A. Bennet and John J. Leonard. A Behavior-Based Approach to Adaptive Feature Detection and Following with Autonomous Underwater Vehicles. *IEEE Journal of Oceanic Engineering*, 25 (2):213–226, April 2000.

Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, April 1986. Also MIT AI Memo 864, September 1985, and republished in Brooks (1999).

Rodney A. Brooks. The Whole Iguana. In Michael Brady, editor, *Robotics Science*, chapter 11, pages 432–456. The MIT Press, 1989.

Rodney A. Brooks. Intelligence Without Reason. MIT AI Lab Memo 1293, April 1991a. Also appeared in Brooks (1999).

Rodney A. Brooks. Intelligence Without Representation. *Artificial Intelligence Journal*, 46:139–159, April 1991b. Also appeared in Brooks (1999).

Rodney A. Brooks. *Cambrian Intelligence: The Early History of the New AI*. MIT Press, Cambridge, MA, 1999.

James Bruce and Manuela Veloso. Real-Time Randomized Path Planning for Robot Navigation. In *Proceedings of IROS-2002*, October 2002.

George B. Dantzig. Programming in a Linear Structure. Comptroller, United States Air Force, February 1948.

Joseph G. Ecker and Michael Kupferschmid. *Introduction to Operations Research*. Krieger Publishing Company, Malabar, FL, 1991.

Barbara Fletcher. UUV Master Plan: A Vision for Navy UUV Development. In *Oceans 2000, MTS/IEEE Conference Proceedings*, Providence RI, September 2000.

Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The Dynamic Window Approach to Collision Avoidance. *IEEE Robotics and Automation*, 4(1), 1997.

Erann Gat. Three-Layer Architectures. In David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter 8, pages 195–210. The AAAI Press / The MIT Press, 1998.

Hiroshi Imai and Takao Asano. Finding the Connected Components and a Maximum Clique of an Intersection Graph of Rectangles in the Plane. *Journal of Algorithms*, 4:310–323, 1983.

Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, New York, NY, 1993.

Oussama Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 500–505, St. Louis, MO, 1985.

Robert Kindel, David Hsu, Jean-Claude Latombe, and Stephan Rock. Kinodynamic Motion Planning Amidst Moving Obstacles, April 2000.

Jana Kosecka and Ruzena Bajcsy. Discrete Event Systems for Autonomous Mobile Agents. *Journal of Robotics and Autonomous Systems*, 12:187–198, 1994.

Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and Alexander schriver, editors. *History of Mathematical Programming: A Collection of Personal Reminiscences*. Elsevier Science Publishers B.V., 1991.

Pattie Maes. The Dynamics of Action Selection. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 991–997, Detroit, MI, 1989.

Pattie Maes. Situated Agents Can Have Goals. In Pattie Maes, editor, *Designing Autonomous Agents*, pages 49–70. MIT/ELSEVIER The MIT Press, 1990.

Maja Mataric. Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior. *Journal of Experimental and Theoretical Artificial Intelligence, special issue on Software architectures for Physical Agents*, 9(2):323–336, 1997.

Kaisa M. Miettinen. *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, Boston, MA, 1999.

Vilfredo Pareto. Cours d'Economie Politique. Libraire Droz, Genève (the first edition in 1896), 1964.

David W. Payton, Julio K. Rosenblatt, and David M. Keirsey. Plan Guided Reaction. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1370–1382, 1990.

Paolo Pirjanian. *Multiple Objective Action Selection & Behavior Fusion*. PhD thesis, Aalborg University, 1998.

Paolo Pirjanian and Maja J. Mataric. Multiple Objective vs. Fuzzy Behavior Coordination. In D. Drainkov and A. Saffiotti, editors, *Lecture Notes in Computer Science on Fuzzy Logic Techniques for Autonomous Vehicle Navigation*, 1999.

John Reif and Zheng Sun. An Efficient Approximation Algorithm for Weighted Region Shortest Path Problem. In *Proceedings of the 4th Workshop on Algorithmic Foundations of Robotics (WAFR2000)*, pages 191–203, March 2000. Published by A. K. Peters Ltd, Hanover, New Hampshire.

John Reif and Zheng Sun. Motion Planning in the Presence of Flows. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures (WADS2001)*, pages 450–461, Brown University, Providence RI, August 2001. Volume 2125 of Lecture Notes in Computer Science.

Jukka Riekki. *Reactive Task Execution of a Mobile Robot*. PhD thesis, Oulu University, 1999.

M. Roeckel, R. Rivoir, and R. Gibson. A Behavior Based Controller Architecture and the Transition to an Industry Application. In *Proceedings of the 1999 - 14 th IEEE International Symposium on Intelligent Control / Intelligent Systems and Semiotics*, Piscataway, NJ, 1999.

Julio K. Rosenblatt. *DAMN: A Distributed Architecture for Mobile Navigation.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1997.

Alessandro Saffiotti, Enrique H. Ruspini, and Kurt Konolige. Using Fuzzy Logic for Mobile Robot Control. In H. J. Zimmerman, editor, *Practical Applications of Fuzzy Technologies*, chapter 5, pages 185–206. Kluwer Academic Publishers, 1999.

Allasandro Saffiotti. The Uses of Fuzzy Logic in Autonomous Robot Navigation: a catalogue raisonné. Technical Report 97-6, IRIDIA, Université Libre de Bruxelles, November 1997.

Zheng Sun and John Reif. BUSHWHACK: An Approximation Algorithm for Minimal Paths Through Pseudo-Euclidean Spaces. In *Proceedings of the 12th Annual International Symposium on Algorithms and Computation (ISAAC01)*, pages 160–171, Christchurch, New Zealand, December 2001. Published in Volume 2223 of Lecture Notes in Computer Science.

Evangelo Triantaphyllou. *Multi-Criteria Decision Making Methods: A Comparative Study.* Kluwer Academic Publishers, 2000.

Robert L. Wernli. Recent U.S. Navy Underwater Vehicle Projects. In *24th UJNR/MFP Meeting*, Honolulu HI, November 2001.

Wayne L. Winston. *Introduction to Mathematical Programming: Applications and Algorithms, 2nd edition.* Duxbury Press, 1995.

John Yen and Nathan Pfluger. A Fuzzy Logic Based Extension to Payton and Rosenblatt's Command Fusion Method for Mobile Robot Navigation. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(6):971–978, June 1995.

Index

IvP function
  accuracy, 34, 36
    capacity, 41
    measurement, 39
    vs. resources, time, 40–41
  creation, 33–37, 113–115
  definition, 21
  uniform distribution, 43–44
IvP model
  strengths, 28, 31
IvP piece
  boundary, 24–25
  combinations, 45–48
  definition, 21
  distribution, 43–44
  interior function, 23–24, 42–43
  intersection, 46
  refinement, 35
  splitting, 35
IvP problem, 25–26
  creation, 33–37
  definition, 25
  solution, 26
IvP problems
  creation, 115–117
  solution, 117
IvP vehicle simulator, *see* vehicle simulator
Arkin (1987), 10
Arkin (1989a), 10
Arkin (1989b), 7
Arkin (1992), 10
Arkin (1998), 2
Benjamin et al. (1993), 1, 89
Bennet and Leonard (2000), 9
Brooks (1986), 2, 6–9
Brooks (1989), 8, 14
Brooks (1991a), 6
Brooks (1999), 6
Bruce and Veloso (2002), 90
Dantzig (1948), 15
Ecker and Kupferschmid (1991), 105
Fletcher (2000), 63
Fox et al. (1997), 78
Gat (1998), 8, 9
Imai and Asano (1983), 57–59
Khatib (1985), 10
Kindel et al. (2000), 90
Kosecka and Bajcsy (1994), 9
Lenstra et al. (1991), 16
Maes (1989), 9
Maes (1990), 7
Miettinen (1999), 17–19, 26, 27
Pareto (1964), 18
Payton et al. (1990), 7
Pirjanian and Mataric (1999), 8, 104
Pirjanian (1998), 2, 8, 9, 66, 90
Reif and Sun (2000), 74
Reif and Sun (2001), 74
Riekki (1999), 2, 3, 12, 66, 100–102
Roeckel et al. (1999), 9
Rosenblatt (1997), 2, 3, 7, 11, 12, 29, 66, 90, 99, 100, 102
Saffiotti et al. (1999), 101–104
Saffiotti (1997), 8

Triantaphyllou (2000), 17
Wernli (2001), 63
Winston (1995), 105, 106
Yen and Pfluger (1995), 101, 103, 104
`nodePiece`
    defined, 46
2-3 tree, *see* plane sweep algorithm

action selection
    complexity tradeoffs, 7
action selection methods, 8–14
    taxonomy, 8, 13
actuators, 22, 66
analytical methods, 31
AUVs, 63–64
    depth, 66
    military AUVs, 63

bathymetry data, 64, 72–74
behavior-based control, 2–3, 5–14
    architecture, 6–8
    control loop, 5, 64
behaviors
    boldest path, 77–78
    quickest path, 77
    safest path, 67–71
    shortest path, 72–77
    steadiest path, 78–79
branch and bound
    integer programming, 107
brute force, 26, 28–31, 55–57
    simplification, 29
    speed, 28

closest point of approach, 67
    determining distance, 68
    determining when, 67–68
    utility metric, 69
CML, 92
combination space, 45–48
Concurrent Mapping and Localization, *see* CML
constraints, 27
control variables
    rate of change, 66
convex programming, 107–109
CPA, *see* closest point of approach

DAMN - Distributed Architecture for Mobile Navigation, 7
decision space, 22–23, 28, 45
    reduced resolution, 30
    resolution, 30
decision variable, 22
    domain, 21, 26
    resolution, 30, 43

feasible region, 26–27
    non-convex, 27
function approximation, 22

global optimality, 21, 26
grid structure, 48–55
    effectiveness, 52
    initializing, 48–49
    intersection information, 49–50
    parameters, 52
    populating, 48–49
    query box, 49, 50
    upper bound information, 50–55

integer programming, 105–107
    example, 106
    NP-complete, 105
interior function, 23–24
    empirical results, 42–43
    intersection, 46
interval programming, *see* IvP
IPAL(), 46–48, 54

leaf node, *see* search tree
loosely coupled problems, 28

mathematical programming, 14–17

MCDM
    decision variables, 17
    feasible space, 17
    general model, 17
    lexicographic method, 18
motion planning
    maneuvering obstacles, 86–89
    moving obstacles, 66, 85–89
moving obstacles, *see* motion planning

Naval Oceanographic Office
    Data Warehouse, 72
node, *see* search tree
non-rectilinear, 24–25
non-rectilinear piece, 21
nondominated solutions, 19
nonlinear programming, 107–109

objective space, 17–19
ownship
    collision avoidance, 67, 77
    control variables, 66
        resolution, 66
    decision space size, 66
    environment, 64–66
    operating depth, 73
    relative distance, 67
    transiting, 77

Pareto optimality, 18–19
piecewise constant, 23–24, 42–43
piecewise linear, 23–24, 42–43
plane sweep algorithm, 57–59
    2-3 tree, 58
    empirical results, 59, 129
    implementation, 58
    limitations, 58–59
preference structure, 26–27

quickest path, *see* behaviors

rectilinear, 21, 24–25
ring functions, 37–39, 111–115
    creation, 111–113
RIPAL(), 46–48, 54–55

safest path, *see* behaviors
search tree, 46–48
    leaf node, 46
    node, 46
    pruning, 48
shortest path
    all sources, 74, 95–97
    behavior, *see* behaviors
    vehicle centric, 74–76
Simultaneous Localization and Mapping, *see* CML
situated agents, 6
solution space, 26
submarine
    maneuver decision aid, 1
subsumption architecture, 6

traveling salesman problem, 106

uniform discrete domain, 22–23
uniform vs. non-uniform pieces, 43–44
unimodal function, 107

value function, 17–19
    explicit vs. implicit, 18
vehicle control loop, 64, 72
vehicle simulator, 81–82, 92–93