An Overview of MOOS-IvP and a Users Guide to the IvP Helm - Release 19.8

Michael R. Benjamin¹, Henrik Schmidt¹, Paul Newman²

¹Department Mechanical Engineering
Massachusetts Institute of Technology, Cambridge MA
²Department of Engineering Science
University of Oxford, Oxford England

August, 2019 - Release 19.8

Last Updated: August 29, 2021

Abstract

This document describes the IvP Helm - an Open Source behavior-based autonomy application for unmanned vehicles. IvP is short for interval programming - a technique for representing and solving multi-objective optimizations problems. Behaviors in the IvP Helm are reconciled using multi-objective optimization when in competition with each other for influence of the vehicle. The IvP Helm is written as a MOOS application where MOOS is a set of Open Source publish-subscribe autonomy middleware tools. This document describes the configuration and use of the IvP Helm, provides examples of simple missions and information on how to download and build the software from the MOOS-IvP server at www.moos-ivp.org.
This work is the product of a multi-year collaboration between the Department of Mechanical Engineering and the Computer Science and Artificial Intelligence Laboratory (CSAIL) at the Massachusetts Institute of Technology in Cambridge Massachusetts, and the Oxford University Mobile Robotics Group.

Points of contact for collaborators:

Dr. Michael R. Benjamin
Department of Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
mikerb@csail.mit.edu

Dr. Paul Newman
Department of Engineering Science
University of Oxford
pnewman@robots.ox.ac.uk

Prof. Henrik Schmidt
Department of Mechanical Engineering
Massachusetts Institute of Technology
henrik@mit.edu

Other collaborators have contributed greatly to the development and testing of software and ideas within, notably - Toby Schneider, Stephanie Kenna, Arjan Vermeij, David Battle, Christian Convey, Chris Gagner.
Sponsorship:
This work is sponsored by Dr. Behzad Kamgar-Parsi and Dr. Don Wagner of the Office of Naval Research (ONR), Code 311. Further support for testing and coursework development sponsored by Battelle, Dr. Robert Carnes.
## Contents

1 Overview of the MOOS-IvP Autonomy Project .................................................. 29
   1.1 Brief Background of MOOS-IvP ............................................................. 29
   1.2 Sponsors of MOOS-IvP .............................................................................. 29
   1.3 Where to Get Further Information ............................................................ 29
       1.3.1 Websites and Email Lists ............................................................... 29
       1.3.2 Documentation ............................................................................. 30

2 Design Considerations of MOOS-IvP ................................................................. 31
   2.1 Public Infrastructure - Layered Capabilities ............................................... 31
   2.2 Code Re-Use ......................................................................................... 32
   2.3 The Backseat Driver Design Philosophy .................................................... 34
   2.4 The Publish-Subscribe Middleware Design Philosophy and MOOS .......... 35
   2.5 The Behavior-Based Control Design Philosophy and IvP Helm .................. 36

3 A Very Brief Overview of MOOS ......................................................................... 38
   3.1 Inter-process communication with Publish/Subscribe ............................... 38
   3.2 Message Content ................................................................................... 38
   3.3 Mail Handling - Publish/Subscribe - in MOOS .......................................... 40
       3.3.1 Publishing Data ............................................................................. 40
       3.3.2 Registering for Notifications ......................................................... 40
       3.3.3 Reading Mail .............................................................................. 40
   3.4 Overloaded Functions in MOOS Applications .............................................. 41
       3.4.1 The Iterate() Method .................................................................. 41
       3.4.2 The OnNewMail() Method ............................................................. 42
       3.4.3 The OnStartUp() Method ............................................................... 42
   3.5 MOOS Mission Configuration Files ............................................................ 42
   3.6 Launching Groups of MOOS Applications with Antler .............................. 43
   3.7 Scoping and Poking the MOOSDB .............................................................. 43
   3.8 A Simple MOOS Application - pXRelay .................................................... 44
       3.8.1 Finding and Launching the pXRelay Example .................................. 45
       3.8.2 Scoping the pXRelay Example with uXMS ..................................... 45
       3.8.3 Seeding the pXRelay Example with the uPokeDB Tool .................... 46
       3.8.4 The pXRelay Example MOOS Configuration File ......................... 47
       3.8.5 Suggestions for Further Things to Try with this Example ................ 48
   3.9 MOOS Applications Available to the Public ................................................. 49
       3.9.1 MOOS Modules from Oxford ......................................................... 49
       3.9.2 Mission Monitoring Modules ......................................................... 50
       3.9.3 Mission Execution Modules ........................................................... 50
       3.9.4 Mission Simulation Modules ........................................................... 51
       3.9.5 Modules for Poking the MOOSDB ................................................ 51
       3.9.6 The Alog Toolbox ....................................................................... 52
       3.9.7 The uField Toolbox ..................................................................... 52

4 A First Example with MOOS-IvP - the Alpha Mission ..................................... 54
4.1 Find and Launch the Alpha Example Mission .................................. 54
4.2 A Look at the Behavior File used in the Alpha Example Mission .......... 56
4.3 A Closer Look at the MOOS Apps in the Alpha Example Mission .......... 58
   4.3.1 Antler and the Antler Configuration Block ............................. 58
   4.3.2 The pMarinePID Application ............................................. 59
   4.3.3 The uSimMarine Application and Configuration Block ................. 59
   4.3.4 The pNodeReporter Application and Configuration Block ............. 60
   4.3.5 The pMarineViewer Application and Configuration Block .......... 60

5 The IvP Helm as a MOOS Application ................................................. 61
   5.1 Overview .................................................................................. 61
   5.2 The Helm State ......................................................................... 62
      5.2.1 Helm State Transitions ....................................................... 62
      5.2.2 What Is and Isn't Happening when the Helm is Parked ............ 63
      5.2.3 Initializing the Helm State at Process Launch Time ................ 64
   5.3 Helm All-Stop Events and All-Stop Status .................................... 64
   5.4 Parameters for the pHelmIvP MOOS Configuration Block ................. 65
      5.4.1 The allow_park Parameter .................................................. 66
      5.4.2 The behaviors Parameter .................................................... 66
      5.4.3 The behavior_dir Parameter ............................................... 66
      5.4.4 The community Parameter .................................................. 66
      5.4.5 The park_on_allstop Parameter .......................................... 66
      5.4.6 The domain Parameter ....................................................... 66
      5.4.7 The ok_skew Parameter ...................................................... 67
      5.4.8 The other_override_var Parameter ..................................... 67
      5.4.9 The start_in_drive Parameter ............................................. 67
      5.4.10 The hold_on_app Parameter .............................................. 67
      5.4.11 The verbose Parameter ..................................................... 67
      5.4.12 An Example pHelmIvP MOOS Configuration Block ............... 68
   5.5 Launching the Helm and Output to the Terminal Window .................. 68
   5.6 Publications and Subscriptions for IvP Helm .................................. 70
      5.6.1 Variables published by the IvP Helm ................................... 70
      5.6.2 Variables Subscribed for by the IvP Helm ............................. 71
   5.7 Using a Standby Helm .............................................................. 72
      5.7.1 Two Types of Helm Failure, the Causes, and Detection ............ 72
      5.7.2 Handling a Helm Crash with the Standby Helm ...................... 73
      5.7.3 Handling a Hung Helm with the Standby Helm ...................... 73
      5.7.4 Activity of the Standby Helm While Standing By .................... 74
      5.7.5 Activity of the Primary Helm After Take-Over ....................... 74
   5.8 Automated Filtering of Successive Duplicate Helm Publications ............ 74
      5.8.1 Motivation for the Duplication Filter .................................. 75
      5.8.2 Implementation and Usage of the Duplication Filter .............. 75
      5.8.3 Clearing the Duplication Filter ......................................... 76

6 IvP Helm Autonomy ................................................................. 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>Behavior Functions Invoked by the Helm</td>
<td>104</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Helm-Invoked Immutable Functions</td>
<td>106</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Helm-Invoked Overloaded Functions</td>
<td>106</td>
</tr>
<tr>
<td>7.5</td>
<td>Local Behavior Utility Functions</td>
<td>107</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Summary of Implementor-Invoked Utility Functions</td>
<td>107</td>
</tr>
<tr>
<td>7.5.2</td>
<td>The Information Buffer</td>
<td>108</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Requesting the Inclusion of a Variable in the Information Buffer</td>
<td>109</td>
</tr>
<tr>
<td>7.5.4</td>
<td>Accessing Variable Information from the Information Buffer</td>
<td>109</td>
</tr>
<tr>
<td>7.6</td>
<td>Overloading the onRunState() and onIdleState() Functions</td>
<td>110</td>
</tr>
<tr>
<td>7.7</td>
<td>Dynamic Behavior Spawning</td>
<td>111</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Behavior Specifications Viewed as Templates</td>
<td>111</td>
</tr>
<tr>
<td>7.7.2</td>
<td>Behavior Completion and Removal from the Helm</td>
<td>112</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Example Missions with Dynamic Behavior Spawning</td>
<td>112</td>
</tr>
<tr>
<td>7.7.4</td>
<td>Examining the Helm’s Life Event History</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>Behaviors of the IvP Helm</td>
<td>114</td>
</tr>
<tr>
<td>9</td>
<td>The Waypoint Behavior</td>
<td>116</td>
</tr>
<tr>
<td>9.1</td>
<td>Configuration Parameters</td>
<td>116</td>
</tr>
<tr>
<td>9.2</td>
<td>Variables Published</td>
<td>119</td>
</tr>
<tr>
<td>9.3</td>
<td>Specifying Waypoints with the points or point Parameter</td>
<td>120</td>
</tr>
<tr>
<td>9.4</td>
<td>The order and repeat Parameters</td>
<td>120</td>
</tr>
<tr>
<td>9.5</td>
<td>The capture_radius and slip_radius Parameters</td>
<td>121</td>
</tr>
<tr>
<td>9.6</td>
<td>The capture_line Parameter</td>
<td>121</td>
</tr>
<tr>
<td>9.7</td>
<td>Track-line Following using the lead, lead_damper, and lead_to_start Parameters</td>
<td>122</td>
</tr>
<tr>
<td>9.8</td>
<td>The wptflag Parameter</td>
<td>123</td>
</tr>
<tr>
<td>9.9</td>
<td>Variables Published by the Waypoint Behavior</td>
<td>123</td>
</tr>
<tr>
<td>9.10</td>
<td>The Objective Function Produced by the Waypoint Behavior</td>
<td>124</td>
</tr>
<tr>
<td>9.11</td>
<td>Further Clarification on the repeat vs. perpetual Parameter</td>
<td>125</td>
</tr>
<tr>
<td>9.12</td>
<td>Visual Hints Defined for the Waypoint Behavior</td>
<td>126</td>
</tr>
<tr>
<td>10</td>
<td>The OpRegion Behavior</td>
<td>127</td>
</tr>
<tr>
<td>10.1</td>
<td>Configuration Parameters</td>
<td>127</td>
</tr>
<tr>
<td>10.2</td>
<td>Variables Published</td>
<td>129</td>
</tr>
<tr>
<td>10.3</td>
<td>Safety Checking Applied to an Operation Region</td>
<td>129</td>
</tr>
<tr>
<td>10.4</td>
<td>Safety Limits on Operation Time, Depth, and Vehicle Altitude</td>
<td>130</td>
</tr>
<tr>
<td>10.5</td>
<td>Variables Published by the OpRegion Behavior</td>
<td>130</td>
</tr>
<tr>
<td>10.6</td>
<td>Publishing Breach Flags</td>
<td>131</td>
</tr>
<tr>
<td>10.7</td>
<td>Resetting a Breach Condition</td>
<td>132</td>
</tr>
<tr>
<td>11</td>
<td>The Loiter Behavior</td>
<td>133</td>
</tr>
<tr>
<td>11.1</td>
<td>Configuration Parameters</td>
<td>133</td>
</tr>
<tr>
<td>11.2</td>
<td>Variables Published</td>
<td>135</td>
</tr>
<tr>
<td>11.3</td>
<td>Detailed Discussions on Loiter Behavior Parameters</td>
<td>136</td>
</tr>
<tr>
<td>11.3.1</td>
<td>The polygon Parameter for Setting the Loiter Region</td>
<td>136</td>
</tr>
<tr>
<td>11.3.2</td>
<td>The updates Parameter for Updating the Loiter Region</td>
<td>136</td>
</tr>
</tbody>
</table>
11.3.3 The `center_activate` Parameter for Using the Current Vehicle Position 137
11.3.4 The `speed` Parameter for Setting the Loiter Speed 137
11.3.5 The `speed_alt` and `use_alt_speed` Parameters 137
11.3.6 The `spiral_factor` Parameter 138
11.3.7 The `acquire_dist` Parameter 138
11.3.8 The `xcenter_assign` and `ycenter_assign` Parameters 138
11.3.9 The `capture_radius` and `slip_radius` Parameters 138
11.3.10 The `post_suffix` Parameter 138
11.3.11 The `clockwise` Parameter for Setting the Loiter Direction 138
11.3.12 The `visual_hints` Parameter 139

11.4 The Loiter Acquisition Mode 140

12 The PeriodicSpeed Behavior 142
12.1 Configuration Parameters 142
12.2 Variables Published 144
12.3 State Transition Policy and Initial Condition Parameters 144

13 The AvoidObstacle Behavior 146
13.1 Overview 146
13.2 The Nature and Origin of Obstacles 147
13.3 Configuration Parameters 147
13.4 Variables Published 149
13.5 Configuring and Using the AvoidObstacle Behavior 150
13.6 Specifying the Behavior Priority Weight Policy 150
13.7 Specifying the Utility Policy of the Behavior 152

14 The PeriodicSurface Behavior 156
14.1 Configuration Parameters 156
14.2 Detailed Discussions of Behavior Parameters 158
14.2.1 The `period` Parameter 158
14.2.2 The `mark_variable` Parameter 158
14.2.3 The `pending_status_var` Parameter 158
14.2.4 The `atsurface_status_var` Parameter 158
14.2.5 The `ascent_speed` Parameter 158
14.2.6 The `ascent_grade` Parameter 158
14.2.7 The `zero_speed_depth` Parameter 159
14.2.8 The `max_time_at_surface` Parameter 159
14.3 Internal States of Periodic Surface Behavior 159

15 The ConstantDepth Behavior 160
15.1 Configuration Parameters 160
15.2 Variables Published 161
15.3 The `duration` Parameter 162
15.4 The ConstantDepth Objective Function 162

16 The ConstantHeading Behavior 163
16.1 Configuration Parameters ........................................... 163
16.2 Variables Published .................................................... 164
16.3 The duration Parameter ............................................. 165
16.4 Behavior Completion .................................................. 165
16.5 The ConstantHeading Objective Function ...................... 165

17 The ConstantSpeed Behavior ........................................... 166
17.1 Configuration Parameters ........................................... 166
17.2 Variables Published .................................................... 167
17.3 The duration Parameter ............................................. 168
17.4 The ConstantSpeed Objective Function ......................... 168

18 The GoToDepth Behavior ............................................... 169
18.1 Configuration Parameters ........................................... 169
18.2 A Detailed Discussion of GoToDepth Behavior Parameters ... 171
18.2.1 The capture_flag Parameter .................................... 171
18.2.2 The capture_delta Parameter .................................. 171
18.2.3 The depth Parameter ............................................. 171
18.2.4 The repeat Parameter ............................................. 171
18.2.5 The perpetual Parameter ........................................ 171

19 The MemoryTurnLimit Behavior ...................................... 173
19.1 Configuration Parameters ........................................... 173
19.2 Calculation of the Heading History .............................. 174
19.3 Variables Published .................................................... 175
19.4 The MemoryTurnLimit Objective Function ..................... 175

20 The StationKeep Behavior .............................................. 176
20.1 Configuration Parameters ........................................... 176
20.2 The station_pt and center_activate Parameters ................ 178
20.3 The swing_time Parameter ......................................... 178
20.4 The inner_radius, outer_radius, and outer_speed Parameters . 179
20.5 Passive Low-Energy Station Keeping Mode .................... 179
20.6 Station Keeping On Demand ........................................ 181
20.7 The visual_hints Parameter ........................................ 182

21 The Timer Behavior ..................................................... 184
21.1 Configuration Parameters ........................................... 184
21.2 Variables Published .................................................... 185

22 The TestFailure Behavior ............................................... 186
22.1 Configuration Parameters ........................................... 186

23 Contact Related Behaviors of the IvP Helm ...................... 189
23.1 Static Versus Dynamically Spawned Behaviors ................. 189
23.2 Exclusion Filters ..................................................... 190
28.5 Visual Preferences .................................................. 235
28.6 Output of the Convoy Behavior in the Form of Publications ......................... 236
  28.6.1 Baked In Publications of the Convoy Behavior .................................. 236
  28.6.2 Event Flag Publications of the Convoy Behavior ................................. 238
  28.6.3 Event Flag Macros of the Convoy Behavior ....................................... 238
28.7 Performance Metrics .................................................. 239
28.8 Configuration Parameters .............................................. 240
28.9 Terms and Definitions ................................................ 242
28.10 Known Shortcomings - Work in Progress .............................................. 243

29 Extended Example Missions with the IvP Helm ........................................... 244
  29.1 Preliminaries .......................................................... 244
  29.2 Using pAntler to Launch Missions ................................................. 244
  29.3 Using the nsplug Utility for Configuring Mission and Behavior Files ............... 244

30 The Charlie Mission ...................................................... 247
  30.1 Launching the Charlie Mission ................................................... 247
  30.2 Topic #1: Hierarchical Mode Declarations in the Charlie Mission ................. 247
  30.3 Topic #2: The Loiter Behavior .................................................. 249
    30.3.1 Loiter Traversal - Clockwise vs. Counter-Clockwise ......................... 249
    30.3.2 Loiter Polygon Shapes, Hexagons, and Ellipses ............................... 250
    30.3.3 Loiter Dynamic Changes to the Polygon Position .............................. 250
    30.3.4 Loiter Robustness to Periodic External Forces .............................. 250
  30.4 Topic #3: The StationKeep Behavior ............................................. 251
    30.4.1 Putting the Vehicle into the Station Keeping Mode ......................... 251
    30.4.2 What is Happening in the Station Keeping Mode ............................. 252
  30.5 Suggestions for Tinkering in the Charlie Mission .................................. 252

31 The Delta Mission ........................................................ 255
  31.1 Overview of the Delta Mission Components and Topics ............................. 255
  31.2 Launching the Delta Mission ................................................... 255
  31.3 Topic #1: Configuring the Helm for Operation at Depth .......................... 256
  31.4 Topic #2: The ConstantDepth Behavior ............................................ 256
  31.5 Topic #3: The PeriodicSurface Behavior .......................................... 257
  31.6 Topic #4: The Waypoint Behavior with Survey Patterns .......................... 258
  31.7 Topic #4: Using pMarineViewer with Geo-referenced Mouse Clicks .............. 259
  31.8 Suggestions for Further Experimenting with the Delta Mission ................... 259
    31.8.1 Failing to Reason about Depth ............................................. 259

32 The Echo Mission ........................................................ 264
  32.1 Overview of the Echo Mission Components and Topics ............................. 264
  32.2 Launching the Echo Mission ................................................... 264
  32.3 Topic #1: The BearingLine Behavior .............................................. 264
  32.4 Topic #2 Dynamic Behavior Spawning .............................................. 264
  32.5 Topic #3: Sending Updates to the Original and Spawned Behaviors ............. 267
  32.6 Suggestions for Tinkering ...................................................... 268

12
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.2.3 The ZAIC LEQ Interface Implementation</td>
<td>301</td>
</tr>
<tr>
<td>37.2.4 Using the ZAIC LEQ Tool</td>
<td>302</td>
</tr>
<tr>
<td>37.2.5 The ZAIC HEQ Tool</td>
<td>303</td>
</tr>
<tr>
<td>37.2.6 A Warning about the Maximum Utility Plateau</td>
<td>304</td>
</tr>
<tr>
<td>37.3 The ZAIC Vector Tool</td>
<td>304</td>
</tr>
<tr>
<td>37.3.1 Brief Overview</td>
<td>304</td>
</tr>
<tr>
<td>37.3.2 Using the ZAIC Vector Tool</td>
<td>305</td>
</tr>
<tr>
<td>38 The Reflector Tool for Building N-Variable IvP Functions</td>
<td>307</td>
</tr>
<tr>
<td>38.1 Overview</td>
<td>307</td>
</tr>
<tr>
<td>38.2 Implementing Underlying Functions within the AOF Class</td>
<td>308</td>
</tr>
<tr>
<td>38.2.1 The AOF Class Definition</td>
<td>308</td>
</tr>
<tr>
<td>38.2.2 An Example Underlying Function Implemented as an AOF Subclass</td>
<td>308</td>
</tr>
<tr>
<td>38.2.3 Another AOF Example Class Implementation for Gaussian Functions</td>
<td>309</td>
</tr>
<tr>
<td>38.3 Basic Reflector Tool Usage Tool with Examples</td>
<td>311</td>
</tr>
<tr>
<td>38.4 The Full Reflector Interface Implementation</td>
<td>314</td>
</tr>
<tr>
<td>39 Optional Advanced Features of the Reflector Tool</td>
<td>316</td>
</tr>
<tr>
<td>39.1 Preliminaries</td>
<td>316</td>
</tr>
<tr>
<td>39.1.1 The Reflector-Script</td>
<td>316</td>
</tr>
<tr>
<td>39.1.2 Specifying a Piece Shape or IvP Domain Point in String Format</td>
<td>316</td>
</tr>
<tr>
<td>39.1.3 Specifying a Region of an IvP Domain in String Format</td>
<td>318</td>
</tr>
<tr>
<td>39.2 Optional Feature #1: Choosing the Piece Shape in Uniform Functions</td>
<td>319</td>
</tr>
<tr>
<td>39.2.1 Potential Advantages</td>
<td>319</td>
</tr>
<tr>
<td>39.2.2 Specifying the Piece Shape Implicitly from a Piece Count Request</td>
<td>319</td>
</tr>
<tr>
<td>39.2.3 Specifying the Uniform Piece Shape Explicitly</td>
<td>321</td>
</tr>
<tr>
<td>39.3 Optional Feature #2: IvP Functions with Directed Refinement</td>
<td>322</td>
</tr>
<tr>
<td>39.4 Optional Feature #3: IvP Functions with Smart Refinement</td>
<td>325</td>
</tr>
<tr>
<td>39.4.1 Potential Advantages</td>
<td>325</td>
</tr>
<tr>
<td>39.4.2 The Smart-Refinement Algorithm</td>
<td>326</td>
</tr>
<tr>
<td>39.4.3 Invoking the Smart-Refine Algorithm in the Reflector</td>
<td>329</td>
</tr>
<tr>
<td>39.5 Optional Feature #4: IvP Functions with Auto-Peak Refinement</td>
<td>330</td>
</tr>
<tr>
<td>39.5.1 Potential Advantages</td>
<td>330</td>
</tr>
<tr>
<td>39.5.2 The Auto-Peak Algorithm</td>
<td>331</td>
</tr>
<tr>
<td>39.5.3 Invoking the Auto-Peak Algorithm in the Reflector</td>
<td>332</td>
</tr>
<tr>
<td>40 An Implementation Example - the SimpleWaypoint Behavior</td>
<td>333</td>
</tr>
<tr>
<td>40.1 The SimpleWaypoint Behavior Class Definition</td>
<td>333</td>
</tr>
<tr>
<td>40.2 The SimpleWaypoint Behavior Class Implementation</td>
<td>334</td>
</tr>
<tr>
<td>40.2.1 The SimpleWaypoint Behavior Constructor</td>
<td>334</td>
</tr>
<tr>
<td>40.2.2 The SimpleWaypoint Behavior setParam() Function</td>
<td>335</td>
</tr>
<tr>
<td>40.2.3 The SimpleWaypoint onIdleState() and postViewPoint() Functions</td>
<td>337</td>
</tr>
<tr>
<td>40.2.4 The SimpleWaypoint Behavior onRunState() Function</td>
<td>337</td>
</tr>
<tr>
<td>40.2.5 The SimpleWaypoint Behavior buildFunctionWithZAIC() Function</td>
<td>340</td>
</tr>
<tr>
<td>40.2.6 The SimpleWaypoint Behavior buildFunctionWithReflector() Function</td>
<td>343</td>
</tr>
</tbody>
</table>
40.3 Running an Example Mission with the SimpleWaypoint Behavior . . . . . . . . . . . . 344

A Use of Logic Expressions 347
B Behavior Summaries 349
C Colors 368
1 Overview of the MOOS-IvP Autonomy Project

1.1 Brief Background of MOOS-IvP

MOOS was written by Paul Newman in 2001 to support operations with autonomous marine vehicles in the MIT Ocean Engineering and the MIT Sea Grant programs. At the time Newman was a post-doc working with John Leonard and has since joined the faculty of the Mobile Robotics Group at Oxford University. MOOS continues to be developed and maintained by Newman at Oxford and the most current version can be found at themoos.org. The MOOS software available in the MOOS-IvP project includes a snapshot of the MOOS code distributed from Oxford. The IvP Helm was developed in 2004 for autonomous control on unmanned marine surface craft, and later underwater platforms. It was written by Mike Benjamin as a post-doc working with John Leonard, and as a research scientist for the Naval Undersea Warfare Center in Newport Rhode Island. The IvP Helm is a single MOOS process that uses multi-objective optimization to implement behavior coordination.

Acronyms

MOOS stands for "Mission Oriented Operating Suite” and its original use was for the Bluefin Odyssey III vehicle owned by MIT. IvP stands for "Interval Programming” which is a mathematical programming model for multi-objective optimization. In the IvP model each objective function is a piecewise linear construct where each piece is an interval in N-Space. The IvP model and algorithms are included in the IvP Helm software as the method for representing and reconciling the output of helm behaviors. The term interval programming was inspired by the mathematical programming models of linear programming (LP) and integer programming (IP). The pseudo-acronym IvP was chosen simply in this spirit and to avoid acronym clashing.

1.2 Sponsors of MOOS-IvP

Original development of MOOS and IvP were more or less infrastructure by-products of other sponsored research in (mostly marine) robotics. Those sponsors were primarily The Office of Naval Research (ONR), as well as the National Oceanic and Atmospheric Administration (NOAA). MOOS and IvP are currently funded by Code 311 at ONR, Dr. Don Wagner and Dr. Behzad Kamgar-Parsi. Testing and development of course work at MIT is further supported by Battelle, Mr. Mike Mellott. The Battelle sponsorship has been very instrumental in the development of the documentation and online course material. MOOS is additionally supported in the U.K. by EPSRC. Early development of IvP benefited from the support of the In-house Laboratory Independent Research (ILIR) program at the Naval Undersea Warfare Center in Newport RI. The ILIR program is funded by ONR.

1.3 Where to Get Further Information

1.3.1 Websites and Email Lists

There are two web sites - the MOOS web site maintained by Oxford University, and the MOOS-IvP web site maintained by MIT. At the time of this writing they are at the following URLs:

• www.themoos.org
What is the difference in content between the two web sites? As discussed previously, MOOS-IvP, as a set of software, refers to the software maintained and distributed from Oxford plus additional MOOS applications including the IvP Helm and library of behaviors. The software bundle released at moos-ivp.org does include the MOOS software from Oxford - usually a particular released version. For the absolute latest in the core MOOS software and documentation on Oxford MOOS modules, the Oxford web site is your source. For the latest on the core IvP Helm, behaviors, and MOOS tools distributed by MIT, the moos-ivp.org web site is the source.

There are two mailing lists open to the public. The first list is for MOOS users, and the second is for MOOS-IvP users. If the topic is related to one of the MOOS modules distributed from the Oxford web site, the proper email list is the "moosusers" mailing list. You can join the "moosusers" mailing list at the following URL:

https://lists.csail.mit.edu/mailman/listinfo/moosusers

For topics related to the IvP Helm or modules distributed on the moos-ivp.org web site that are not part of the Oxford MOOS distribution (see the software page on moos-ivp.org for help in drawing the distinction), the "moosivp" mailing list is appropriate. You can join the "moosivp" mailing list at the following URL:

https://lists.csail.mit.edu/mailman/listinfo/moosivp

1.3.2 Documentation

Documentation on MOOS can be found on the MOOS web site:

www.themoos.org

This includes documentation on the MOOS architecture, programming new MOOS applications as well as documentation on several bread-and-butter applications such as pAntler, pLogger, uMS, pShare, iRemote, iMatlab, pScheduler and more. Documentation on the IvP Helm, behaviors and autonomy related MOOS applications not from Oxford can be found on the www.moos-ivp.org web site under the Documentation link.
2 Design Considerations of MOOS-IvP

The primary motivation in the design of MOOS-IvP is to build highly capable autonomous systems. Part of this picture includes doing so at a reduced short and long-term cost and a reduced time line. By "design" we mean both the choice in architectures and algorithms as well as the choice to make key modules for infrastructure, basic autonomy and advanced tools available to the public under an Open Source license. The MOOS-IvP software design is based on three architecture philosophies, (a) the backseat driver paradigm, (b) publish and subscribe autonomy middleware, and (c) behavior based autonomy. The common thread is the ability to separate the development of software for an overall system into distinct modules coordinated by infrastructure software made available to the public domain.

2.1 Public Infrastructure - Layered Capabilities

The central architecture idea of both MOOS and IvP is the separation of overall capability into separate and distinct modules. The unique contributions of MOOS and IvP are the methods used to coordinate those modules. A second central idea is the decision to make algorithms and software modules for infrastructure, basic autonomy and advanced tools available to the public under an Open Source license. The idea is pictured in Figure 1. There are three things in this picture - (a) modules that actually perform a function (the wedges), (b) modules that coordinate other modules (the center of the wheel), and (c) standard wrapper software use by each module to allow it to be coordinated (the spokes).

Figure 1: Public Infrastructure - Layered Capabilities: The center of the wheel represents MOOS-IvP Core. For MOOS this means the MOOSDB and the message passing and scheduling algorithms. For IvP this means the IvP helm behavior management and the multi-objective optimization solver. The wedges on the wheel represent individual modules - either MOOS processes or IvP behaviors. The spokes of the wheel represent the idea that each module inherits from a superclass to grab functionality key to plugging into the core. Each wedge or module contains a wrapper defined by the superclass that augments the function of the individual module. The darker wedges indicate publicly available modules and the lighter ones are modules added by users to augment the public set to comprise a particular fielded autonomy system.

The darker wedges in Figure 1 represent application modules (not infrastructure) that provide basic functionality and are publicly available. However, they do not hold any special immutable status. They can be replaced with a better version, or, since the source code is available, the code of
the existing module can be changed or augmented to provide a better or different version (hopefully with a different name - see the section on branching below). Later sections provide an overview of about 40 or so particular modules that are currently available. By modules we mean MOOS applications and IvP behaviors and the above comments hold in either case. The yellow wedges in Figure 1 represent the imaginable unimplemented modules or functionality. A particular fielded MOOS-IvP autonomy system typically is comprised of (a) the MOOS-IvP core modules, (b) some of the publicly available MOOS applications and IvP behaviors, and (c) additional perhaps non-public MOOS applications and IvP behaviors provided by one or more third party developers.

The objective of the public-infrastructure/layered-capabilities idea is to strike an important balance - the balance between effective code re-use and the need for users to retain privacy regarding how they choose to augment the public codebase with modules of their own to realize a particular autonomy system. The benefits of code re-use are an important motivation in fundamental architecture decisions in both MOOS and IvP. The modules that comprise the public MOOS-IvP codebase described in this document represent over twenty work-years of development effort. Furthermore, certain core components of the codebase have had hundreds if not thousands of hours of usage on a dozen or so fielded platform types in a variety of situations. The issue of code re-use is discussed next.

2.2 Code Re-Use

Code re-use is critical, and starts with the ability to have a system comprised of separate but coordinated modules. They key technical hurdle is to achieve module separation without invoking a substantial hit on performance. In short, MOOS middleware is a way of coordinating separate processes running on a single computer or over several networked computers. IvP is a way of coordinating several autonomy behaviors running within a single MOOS process.

Factors Contributing to Code Re-use:

- **Freedom from proprietary issues.** Software serving as infrastructure shared by all components (MOOS processes and IvP behaviors) are available under an Open Source license. In addition many mature MOOS and IvP modules providing commonly needed capabilities are also publicly available. Proprietary or non-publicly released code may certainly co-exist with non-proprietary public code to comprise a larger autonomy system. Such a system would retain a strategic edge over competitors if desired, but have a subset of components common with other users.

- **Module independence.** Maintaining or augmenting a system comprised of a set of distinct modules can begin to break down if modules are not independent with simple easy-to-augment interfaces. Compile dependencies between modules need to be minimized or eliminated. The maintenance of core software libraries and application code should be decoupled completely from the issues of 3rd party additional code.

- **Simple well-documented interfaces.** The effort required to add modules to the code base should be minimized. Documentation is needed for both (a) using the publicly available applications and libraries, and (b) guiding users in adding their own modules.

- **Freedom to innovate.** The infrastructure does not put undue restrictions on how basic problems
can be solved. The infrastructure remains agnostic to techniques and algorithms used in the modules. No module is sacred and any module may be replaced.

Benefits of Code Re-Use:

- **Diversity of contributors.** Increasingly, an autonomy system contains many components that touch many areas of expertise. This would be true even for a vanilla use of a vehicle, but is compounded when considering the variety of sensors and missions and ways of exploiting sensors in achieving mission objectives. A system that allows for wide code re-use is also a system that allows module contributions from a wide set of developers or experts. This has a substantial impact on the issues mentioned below of lower cost, higher quality and reliability, and reduced development time line.

- **Lower cost.** One immediate benefit of code re-use is the avoidance of repeatedly re-inventing modules. A group can build capabilities incrementally and experts are free to concentrate on their area and develop only the modules that reflect their skill set and interests. Perhaps more important, code re-use gives the systems integrator choices in building a complete system from individual modules. Having choices leads to increased leverage in bargaining for favorable licensing terms or even non-proprietary terms for a new module. Favorable licensing terms arranged at the outset can lead to substantially lower long-term costs for future code maintenance or augmentation of software.

- **Higher performance capability.** Code re-use enhances performance capability in two ways. First, since experts are free to be experts without re-inventing the modules outside their expertise and provided by others, their own work is more likely to be more focused and efficient. They are likely to achieve a higher capability for a given a finite investment and given finite performance time. Second, since code re-use gives a systems integrator choices, this creates a meritocracy based on optimal performance-cost ratio of candidate software modules. The under-capable, more expensive module is less likely to diminish the overall autonomy capability if an alternative module is developed to offer a competitive choice. Survival of the fittest.

- **Higher performance reliability.** An important part of system reliability is testing. The more testing time and the greater diversity of testing scenarios the better. And of course the more time spent testing on physical vehicles versus simulation the better. By making core components of a codebase public and permitting re-use by a community of users, that community provides back an enormous service by simply using the software and complaining when or if something goes wrong. Certain core components of the MOOS-IvP codebase have had hundreds if not thousands of hours of usage on a dozen or so platform types in a variety of situations. And many more hours in simulation. Testing doesn’t replace good coding practice or formal methods for testing and verifying correctness, but it complements those two aspects and is enhanced by code re-use.

- **Reduced development time line.** Code re-use means less code is being re-developed which leads to quicker overall system development. More subtly, since code re-use can provide a systems integrator choices and competition on individual modules, development time can be reduced as a consequent. An integrator may simply accept the module developed the quickest,
or the competition itself may speed up development. If choices and competition result in more favorable license agreements between the integrator and developer, this in itself may streamline agreements for code maintenance and augmentation in the long term. Finally, as discussed above, if code re-use leads to an element of community-driven bug testing, this will also quicken the pace in the evolution toward a mature and reliable autonomy system.

2.3 The Backseat Driver Design Philosophy

The key idea in the backseat driver paradigm is the separation between vehicle control and vehicle autonomy. The vehicle control system runs on a platform’s main vehicle computer and the autonomy system runs on a separate payload computer. This separation is also referred to as the mission controller - vehicle controller interface. A primary benefit is the decoupling of the platform autonomy system from the actual vehicle hardware. The vehicle manufacturer provides a navigation and control system capable of streaming vehicle position and trajectory information from the main vehicle computer, and accepting a stream of autonomy decisions such as heading, speed and depth in return from the payload computer. Exactly how the vehicle navigates and implements control is largely unspecified to the autonomy system running in the payload. The relationship is depicted in Figure 2.

Figure 2: The backseat driver paradigm: The key idea is the separation of vehicle autonomy from vehicle control. The autonomy system provides heading, speed and depth commands to the vehicle control system. The vehicle control system executes the control and passes navigation information, e.g., position, heading and speed, to the autonomy system. The backseat paradigm is agnostic regarding how the autonomy system implemented, but in this figure the MOOS-IvP autonomy architecture is depicted.

The autonomy system on the payload computer consists of a set of distinct processes communicating through a publish-subscribe database called the MOOSDB (Mission Oriented Operating Suite - Database). One such process is an interface to the main vehicle computer, and another key process is the IvP Helm implementing the behavior-based autonomy system. The MOOS community is referred to as the "larger autonomy" system, or the "autonomy system as a whole" since MOOS itself is middleware, and actual autonomous decision making, sensor processing, contact management
etc., are implemented as individual MOOS processes.

2.4 The Publish-Subscribe Middleware Design Philosophy and MOOS

MOOS provides a middleware capability based on the publish-subscribe architecture and protocol. Each process communicates with each other through a single database process in a star topology (Figure 3). The interface of a particular process is described by what messages it produces (publications) and what messages it consumes (subscriptions). Each message is a simple variable-value pair where the values are typically either string or numerical values such as (STATE, "DEPLOY"), or (NAV_SPEED, 2.2). MOOS messages may also contain raw binary data for passing images for example.

![Figure 3: A MOOS community:](image)

The key idea with respect to facilitating code re-use is that applications are largely independent, defined only by their interface, and any application is easily replaceable with an improved version with a matching interface. Since MOOS Core and many common applications are publicly available along with source code under an Open Source GPL license, a user may develop an improved module by altering existing source code and introduce a new version under a different name. With MOOS-IvP 13.2 which includes MOOS V10, the MOOS libraries are distributed under an LGPL license, to allow the development and use of commercial MOOS applications alongside open source applications. The MOOSDB and MOOS applications remain under an GPL license. The term MOOS Core refers to (a) the MOOSDB application, and (b) the MOOS Application superclass that each individual MOOS application inherits from to allow connectivity to a running MOOSDB. Holding the MOOS Core part of the codebase constant between MOOS developers enables the plug-and-play nature of applications.
2.5 The Behavior-Based Control Design Philosophy and IvP Helm

The IvP Helm runs as a single MOOS application and uses a behavior-based architecture for implementing autonomy. Behaviors are distinct software modules that can be described as self-contained mini expert systems dedicated to a particular aspect of overall vehicle autonomy. The helm implementation and each behavior implementation exposes an interface for configuration by the user for a particular set of missions. This configuration often contains particulars such as a certain set of waypoints, search area, vehicle speed, and so on. It also contains a specification of state spaces that determine which behaviors are active under what situations, and how states are transitioned. When multiple behaviors are active and competing for influence of the vehicle, the IvP solver is used to reconcile the behaviors (Figure 4).

The IvP solver performs multi-objective optimization on the set of functions to find the single best vehicle action, which is then published to the MOOSDB. The functions are built and the set is solved on each iteration of the helm - typically one to four times per second. Only a subset of behaviors are active at any given time depending on the vehicle situation, and the state space configuration provided by the user.

The concept of a behavior-based architecture is often attributed to [38]. Since then various solutions to the issue of action selection, i.e., the issue of coordinating competing behaviors, have been put forth and implemented in physical systems. The simplest approach is to prioritize behaviors in a way that the highest priority behavior locks out all others as in the Subsumption Architecture in [38]. Another approach is referred to as the potential fields, or vector summation approach (See [1], [43])
which considers the average action between multiple behaviors to be a reasonable compromise. These action-selection approaches have been used with reasonable effectiveness on a variety of platforms, including indoor robots, e.g., [1], [2], [49], [51], land vehicles, e.g., [52], and marine vehicles, e.g., [37], [39], [44], [54], [55]. However, action-selection via the identification of a single highest priority behavior and via vector summation have well known shortcomings later described in [49], [51] and [52] in which the authors advocated for the use of multi-objective optimization as a more suitable, although more computationally expensive, method for action selection. The IvP model is a method for implementing multi-objective function based action-selection that is computationally viable in the IvP Helm implementation.
3 A Very Brief Overview of MOOS

MOOS is often described as autonomy *middleware* which implies that it is a kind of glue that connects a collection of applications where the real work happens. MOOS does indeed connect a collection of applications, of which the IvP Helm is one. MOOS is cross platform stand-alone and dependency free. It needs no other third-party libraries. Each application inherits a generic MOOS interface whose implementation provides a powerful, easy-to-use means of communicating with other applications and controlling the relative frequency at which the application executes its primary set of functions. Due to its combination of ease-of-use, general extendibility and reliability, it has been used in the classroom by students with no prior experience, as well as on many extended field exercises with substantial robotic resources at stake. To frame the later discussion of the IvP Helm, the basic issues regarding MOOS applications are introduced here. For further information on the original design of MOOS see [47].

3.1 Inter-process communication with Publish/Subscribe

MOOS has a star-like topology as depicted in Figure 3. Application within a MOOS community (a MOOSApp) have a connection to a single MOOS Database (called MOOSDB) lying at the heart of the software suite. All communication happens via this central server application. The network has the following properties:

- No peer-to-peer communication.
- Communication between the client and server is initiated by the client, i.e., the MOOSDB never makes a unsolicited attempt to contact a MOOSApp from out of the blue
- Each client has a unique name.
- A given client need have no knowledge of what other clients exist.
- One client does not transmit data to another - it can only be sent to the MOOSDB and from there to other clients. Modern versions of the library sport a sub-one millisecond latency when transporting multi-MB payloads between processes.
- The star network can be distributed over any number of machines running any combination of supported operating systems.
- The communications layer supports clock synchronization across all connected clients and in the same vein, can support "time acceleration" whereby all connected clients operate in an accelerated time stream - something that is very useful in simulations involving many processes distributed over many machines.
- data can be sent in small bites as "string" or "double" packets or in arbitrarily large binary packets.

3.2 Message Content

The communications API in MOOS allows data to be transmitted between the MOOSDB and a client. The meaning of that data is dependent on the role of the client. However the form of that data is not constrained by MOOS although for the sake on convenience MOOS does offer bespoke support for small "double" and string payloads. (Note that very early versions of MOOS only
allowed data to be sent as strings or doubles - but this restriction is now long gone.) Data is packed into messages which contain other salient information shown in Table 1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the data</td>
</tr>
<tr>
<td>String Value</td>
<td>Data in string format</td>
</tr>
<tr>
<td>Double Value</td>
<td>Numeric double float data</td>
</tr>
<tr>
<td>Source</td>
<td>Name of client that sent this data to the MOOSDB</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>Supplemental message information, e.g., IVP behavior source</td>
</tr>
<tr>
<td>Time</td>
<td>Time at which the data was written</td>
</tr>
<tr>
<td>Data Type</td>
<td>Type of data (STRING or DOUBLE or BINARY)</td>
</tr>
<tr>
<td>Message Type</td>
<td>Type of Message (usually NOTIFICATION)</td>
</tr>
<tr>
<td>Source Community</td>
<td>The community to which the source process belongs</td>
</tr>
</tbody>
</table>

Table 1: The contents of MOOS message.

Often it is convenient to send data in string format for example the string "Type=EST,Name=AUV,Pos=[3x1]3.4,6.3" might describe the position estimate of a vehicle called "AUV" as a 3x1 column vector. This is human readable and does not require the sharing and synchronizing of header files to ensure both sender and recipient understand how to interpret data (as is the case with binary data). It is quite common for MOOS applications to communicate with string data in a concatenation of comma separated "name=value" pairs.

- Strings are human readable.
- All data becomes the same type.
- Logging files are human readable (they can be compressed for storage).
- Replaying a log file is simply a case of reading strings from a file and "throwing" them back at the MOOSDB in time order.
- The contents and internal order of strings transmitted by an application can be changed without the need to recompile consumers (subscribers to that data) - users simply would not understand new data fields but they would not crash.

The above are well understood benefits of sending self-explanatory ASCII data. However many applications use data types which do not lend themselves to verbose serialization to strings — think for example about camera image data being generated at 40Hz in full colour. At this point the need to send binary data is clear and of course MOOS supports it transparently (and the application pLogger supports logging and replaying it).

At this point it is up to the user to ensure that the binary data can be interpreted by all clients and that any and all perturbations to the data structures are distributed and compiled into each and every client. It is here that modern serialization tools such as "Google Protocol Buffers" find application. They offer a seamless way to serialize complex data structures into binary streams. Crucially they offer forward compatibility – it is possible to update and augment data structures with new fields in the comforting knowledge that all existing apps will still be able to interpret the data - they just won’t parse the new additions.
3.3 Mail Handling - Publish/Subscribe - in MOOS

Each MOOS application is a client having a connection to the MOOSDB. This connection is made on the client side and the client manages a threaded machinery that coordinates the communication with the MOOSDB. This completely hides the intricacies and timings of the communications from the rest of the application and provides a small, well defined set of methods to handle data transfer. The application can:

1. Publish data - issue a notification on named data.
2. Register for notifications on named data.
3. Collect notifications on named data - reading mail.

3.3.1 Publishing Data

Data is published as a pair - a variable and value - that constitute the heart of a MOOS message described in Table 1. The client invokes the `Notify(VarName, VarValue)` command where appropriate in the client code. The above command is implemented both for string, double and binary values, and the rest of the fields described in Table 1 are filled in automatically. Each notification results in another entry in the client’s ”outbox”, which in older versions of MOOS, is emptied the next time the **MOOSDB** accepts an incoming call from the client or in recent versions, is pushed instantaneously to all interested clients.

3.3.2 Registering for Notifications

Assume that a list of names of data published has been provided by the author of a particular MOOS application. For example, a application that interfaces to a GPS sensor may publish data called `GPS_X` and `GPS_Y`. A different application may register its interest in this data by subscribing or registering for it. An application can register for notifications using a single method `Register()` specifying both the name of the data and the maximum rate at which the client would like to be informed that the data has been changed. The latter parameter is specified in terms of the minimum possible time between notifications for a named variable. For example setting it to zero would result in the client receiving each and every change notification issued on that variable. MOOS V10 and later also supports ”wildcard” subscriptions. For example a client can register for "*:::*" to receive all messages from all other clients. Or "GPS_:?:NAV" to receive messages beginning with "GPS_" from any process with a four letter name ending in "NAV".

3.3.3 Reading Mail

A client can enquire at any time whether it has received any new notifications from the **MOOSDB** by invoking the `Fetch` method. The function fills in a list of notification messages with the fields given in Table 1. Note that a single call to `Fetch` may result in being presented with several notifications corresponding to the same named data. This implies that several changes were made to the data since the last client-server conversation. However, the time difference between these similar messages will never be less than that specified in the `Register()` function described above. In typical applications the `Fetch` command is called on the client’s behalf just prior to the `Iterate()` method, and the messages are handled in the user overloaded `OnNewMail()` method.

27
3.4 Overloaded Functions in MOOS Applications

MOOS provides a base class called `CMOOSApp` which simplifies the writing of a new MOOS application as a derived subclass. Beneath the hood of the `CMOOSApp` class is a loop which repetitively calls a function called `Iterate()` which by default does nothing. One of the jobs as a writer of a new MOOS-enabled application is to flesh this function out with the code that makes the application do what we want. Behind the scenes this overall loop in `CMOOSApp` is also checking to see if new data has been delivered to the application. If it has, another virtual function, `OnNewMail()`, is called. This is the function within which code is written to process the newly delivered data.

![Diagram of CMOOSApp::Run()](image)

Figure 5: **Key virtual functions of the MOOS application base class**: The flow of execution once `Run()` has been called on a class derived from `CMOOSApp`. The scrolls indicate where users of the functionality of `CMOOSApp` will be writing new code that implements whatever it is that is wanted from the new applications. Note that it is not the case (as the above may suggest) that mail is polled for - in modern incantations of MOOS it is pushed to a client a synchronously `OnNewMail()` is called as soon as `Iterate()` is not running.

The roles of the three virtual functions in Figure 5 are discussed below. The `pHelmIvP` application does indeed inherit from `CMOOSApp` and overload these functions. The base class contains other virtual functions (`OnConnectToServer()` and `OnDisconnectFromServer()`) discussed in [47].

### 3.4.1 The `Iterate()` Method

By overriding the `CMOOSApp::Iterate()` function in a new derived class, the author creates a function from which the work that the application is tasked with doing can be orchestrated. In the `pHelmIvP` application, this method will consider the next best vehicle decision, typically in the form of deciding values for the vehicle heading, speed and depth. The rate at which `Iterate()` is called by the `SetAppFreq()` method or by specifying the `AppTick` parameter in a mission file. Note that the requested frequency specifies the maximum frequency at which `Iterate()` will be called - it does not guarantee that it will be called at the requested rate. For example if you write code in `Iterate()` that takes 1 second to complete there is no way that this method can be called at more than 1Hz. If you want to call `Iterate()` as fast as is possible simply request a frequency of zero - but you may want to reconsider why you need such a greedy application.
3.4.2 The OnNewMail() Method

Just before Iterate() is called, the CMOOSApp base class determines whether new mail is present, i.e., whether some other process has posted data for which the client has previously registered, as described above. If new mail is waiting, the CMOOSApp base class calls the OnNewMail() virtual function, typically overloaded by the application. The mail arrives in the form of a list of CMOOSMsg objects (see Table 1). The programmer is free to iterate over this collection examining who sent the data, what it pertains to, how old it is, whether or not it is string or numerical data and to act on or process the data accordingly. In recent versions of MOOS it is possible to have OnNewMail() called in a directly and rapidly in response to new mail being received by the back-end communications threads. This architecture allows for very rapid response times (sub ms) between a client posting data and it being received and handled by all interested parties.

3.4.3 The OnStartUp() Method

The OnStartUp() function is called just before the application enters into its own forever-loop depicted in Figure 5. This is the application that implements the application’s initialization code, and in particular reads configuration parameters (including those that modify the default behavior of the CMOOSApp base class) from a file.

3.5 MOOS Mission Configuration Files

Every MOOS process can read configuration parameters from a mission file which by convention has a .moos extension. Traditionally MOOS processes share the same mission file to the maximum extent possible. For example, it is customary for there to be one common mission file for all MOOS processes running on a given machine. Every MOOS process has information contained in a configuration block within a *.moos file. The block begins with the statement

\[
\text{ProcessConfig = ProcessName}
\]

where ProcessName is the unique name the application will use when connecting to the MOOSDB. The configuration block is delimited by braces. Within the braces there is a collection of parameter statements, one per line. Each statement is written as:

\[
\text{ParameterName = value}
\]

where value can be any string or numeric value. All applications deriving from CMOOSApp inherit several important configuration options. The most important options for CMOOSApp derived applications are CommsTick and AppTick. The former configures how often the communications thread talks to the MOOSDB and the latter how often (approximately) Iterate() will be called.

Parameters may also be defined at the "global" level, i.e., not in any particular process’ configuration block. Three parameters that are mandatory and typically found at the top of all *.moos files are: ServerHost naming the IP address associated with the MOOSDB server being launched with this file, ServerPort naming the port number over which the MOOSDB server is communicating with clients, and Community naming the community comprising the server and clients. An example is shown in lines 1-3 in Listing 4.

29
3.6 Launching Groups of MOOS Applications with Antler

Antler provides a simple and compact way to start a MOOS mission comprised of several MOOS processes, a.k.a., a MOOS community. For example if the desired mission file is `alpha.moos` then executing the following from a terminal shell:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ pAntler alpha.moos
```

will launch the required processes for the mission. It reads from its configuration block (which is declared as `ProcessConfig=ANTLER`) a list of process names that will constitute the MOOS community. Each process to be launched is specified with a line with the general syntax

where `LaunchConfiguration` is an optional comma-separated list of `parameter=value` pairs which collectively control how the process `procname` (for example `pHelmIvP`, or `pLogger` or `MOOSDB`) is launched. Exactly what parameters can be specified is outside the scope of this discussion. Antler looks through its entire configuration block and launches one process for every line which begins with the `RUN=` left-hand side. When all processes have been launched Antler waits for all of them to exit and then quits itself.

There are many more aspects of Antler not discussed here but can be found in the Antler documentation at the MOOS web site (see Section 1.3). These include hooks for altering the console appearance for each launched process, controlling the search path for specifying how executables are located on the host file system, passing parameters to launched processes, running multiple instances of a particular process, and using Antler to launch multiple distinct communities on a network.

3.7 Scoping and Poking the MOOSDB

An important tool for writing and debugging MOOS applications (and IvP Helm behaviors) is the ability for the user to interact with an active MOOS community and see the current values of particular MOOS variables (scoping the DB) and to alter one or more variables with a desired value (poking the DB). Below are listed tools for scoping and poking respectively. More information on each can be found on the Oxford or MIT web sites, or in in some instances, other parts of this document.

Tools for scoping the MOOSDB:

- **uMS** - A GUI-based tool written in FLTK and maintained and distributed from the Oxford website.
- **uXMS** - A terminal-based tool maintained and distributed from the MIT website
- **uHelmScope** - A terminal-based tool specialized for displaying information about a running instance of the helm, but it also contains a general-purpose scoping utility similar to uXMS. Distributed from the MIT website.
Tools for poking the MOOSDB:

- **uMS** - The GUI-based tool for scoping, listed above, also provides a means for poking. Distributed from the Oxford website.
- **uPokeDB** - A light-weight command-line tool for poking one or more variable-value pairs, with the option of scoping on the before and after values of the poked variable before exiting. Distributed from the MIT website.
- **pMarineViewer** - A GUI-based tool primarily used for rendering the paths of vehicles in 2D space on a Geo display, but also can be configured to poke the DB with variable-value pairs connected to buttons on the display. Distributed from the MIT website.
- **uTimerScript** - Allows the user to script a set of pre-configured pokes to a MOOSDB with each entry in the script happening after a specified amount of time. Script may be paused or fast-forwarded. Events may also be configured with random values and happen randomly in a chosen window of time. Distributed from the MIT website.
- **uTermCommand** - A terminal-based tool for poking the DB with pre-defined variable-value pairs. The user can configure the tool to associate aliases (as short as a single character) to quickly poke the DB. Distributed from the MIT website.
- **iRemote** - A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associate a pre-defined variable-value poke with any un-mapped key on the keyboard. Distributed from the Oxford website.

The above list is almost certainly not a complete list for scoping and poking a MOOSDB, but it’s a decent start.

### 3.8 A Simple MOOS Application - pXRelay

The bundle of applications distributed from [www.moos-ivp.org](http://www.moos-ivp.org) contains a very simple MOOS application called **pXRelay**. The pXRelay application registers for a single “input” MOOS variable and publishes a single “output” MOOS variable. It makes a single publication on the output variable for each mail message received on the input variable. The value published is simply a counter representing the number of times the variable has been published. By running two (differently named) versions of pXRelay with complementary input/output variables, the two processes will perpetuate some basic publish/subscribe handshaking. This application is distributed primarily as a simple example of a MOOS application that allows for some illustration of the following topics introduced up to this point:

- Finding and launching with pAntler example code distributed with the MOOS-IvP software bundle.
- An example mission configuration file.
- Scoping variables on a running MOOSDB with the uXMS tool.
- Poking the MOOSDB with variable-value pairs using the uPokeDB tool.
- Illustrating the OnStartUp(), OnNewMail(), and Iterate() overloaded functions of the CMOOSApp base class.
Besides touching on these topics, the collection of files in the pXRelay source code sub-directory is not a bad template from which to build your own modules.

### 3.8.1 Finding and Launching the pXRelay Example

The pXRelay example mission should be in the same directory tree containing the source code. There is a single mission file, xrelay.moos:

```
moos-ivp/
 MOOS/
 ivp/
 missions/
  xrelay/
   xrelay.moos  ---- The MOOS file
```

To run this mission from a terminal window, simply change directories and launch:

```
$ cd moos-ivp/ivp/missions/xrelay
$ pAntler xrelay.moos
```

After pAntler has launched each process, there should be four open terminal windows, one for each pXRelay process, one for uXMS, and one for the MOOSDB itself.

### 3.8.2 Scoping the pXRelay Example with uXMS

Among the four windows launched in the example, the window to watch is the uXMS window, which should have output similar to the following (minus the line numbers):

[Listing 3.1: Example uXMS output after the pXRelay example is launched.]

<table>
<thead>
<tr>
<th></th>
<th>VarName</th>
<th>Source</th>
<th>Time</th>
<th>Community</th>
<th>VarValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>------------</td>
<td>------------</td>
<td>-------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>1</td>
<td>------------</td>
<td>------------</td>
<td>-------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>2</td>
<td>APPLES</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>3</td>
<td>PEARs</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>4</td>
<td>APPLES_ITER_HZ</td>
<td>pXRelay_APPLES</td>
<td>14.93</td>
<td>xrelay</td>
<td>24.93561</td>
</tr>
<tr>
<td>5</td>
<td>PEARs_ITER_HZ</td>
<td>pXRelay_PEARs</td>
<td>14.94</td>
<td>xrelay</td>
<td>24.93683</td>
</tr>
<tr>
<td>6</td>
<td>APPLES_POST_HZ</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>7</td>
<td>PEARs_POST_HZ</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Initially the only thing that is changing in this window is the integer at the end of line 1 representing the number of updates written to the terminal. Here uXMS is configured to scope on the six variables shown in the VarName column. Column 2 shows which process last posted on the variable, column 3 shows when the last posting occurred, column 4 shows the community name from which the post originated, and column 5 shows the current value of the variable. The "n/a" entries indicate that a process has yet to write to the given variable. For further info on the workings of uXMS see [31], or type 'h' to see the help menu.

There are two pXRelay processes running - one under the alias pXRelay_APPLES publishing the variable APPLES as its output variable, APPLES_ITER_HZ indicating the frequency in which the Iterate() function is executed, and APPLES_POST_HZ indicating the frequency at which the output variable is posted. There is likewise a pXRelay_PEARs process and the corresponding output variables.
3.8.3 Seeding the pXRelay Example with the uPokeDB Tool

Upon launching the pXRelay example, the only variables actively changing are the \_*\_ITER\_HZ\_*\_variables (lines 4-5 in Listing 1) which confirm that the Iterate() loop in each process is indeed being executed. The output for the other variables in Listing 1 reflect the fact that the two processes have not yet begun handshaking. This can be kicked off by poking the APPLES (or PEAR\_S) variable, which is the input variable for pXRelay\_PEAR\_S, by typing the following:

```bash
$ cd moos-ivp/ivp/missions/xrelay
$ uPokeDB xrelay.moos APPLES=1
```

The uPokeDB tool will publish to the MOOSDB the given variable-value pair APPLES=1. It also takes as an argument the mission file, xrelay.moos, to read information on where the MOOSDB is running in terms of machine name and port number. The output should look similar to the following:

Listing 3.2: Example uPokeDB output after poking the MOOSDB with APPLES=1.

<table>
<thead>
<tr>
<th>VarName</th>
<th>Source</th>
<th>Time</th>
<th>VarValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>40.19</td>
<td>1.00000&quot;</td>
</tr>
</tbody>
</table>

The output of uPokeDB first shows the value of the variable prior to the poke, and then the value afterwards. Further information on the uPokeDB tool can be found in [31]. Once the MOOSDB has been poked as above, the pXRelay\_PEAR\_S application will receive this mail and, in return, will write to its output variable PEAR\_S, which in turn will be read by pXRelay\_APPLE\_S and the two processes will continue thereafter to write and read their input and output variables. This progression can be observed in the uXMS terminal, which may look something like that shown in Listing 3:

Listing 3.3: Example uXMS output after the pXRelay example is seeded.

<table>
<thead>
<tr>
<th>VarName</th>
<th>Source</th>
<th>Time</th>
<th>Community</th>
<th>VarValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLES</td>
<td>pXRelay_APPLE_S</td>
<td>44.78</td>
<td>xrelay</td>
<td>151</td>
</tr>
<tr>
<td>PEAR_S</td>
<td>pXRelay_PEAR_S</td>
<td>44.74</td>
<td>xrelay</td>
<td>151</td>
</tr>
<tr>
<td>APPLES_ITER_HZ</td>
<td>pXRelay_PEAR_S</td>
<td>44.7</td>
<td>xrelay</td>
<td>24.90495</td>
</tr>
<tr>
<td>PEAR_S_ITER_HZ</td>
<td>pXRelay_PEAR_S</td>
<td>44.7</td>
<td>xrelay</td>
<td>24.90427</td>
</tr>
<tr>
<td>APPLES_POST_HZ</td>
<td>pXRelay_APPLE_S</td>
<td>44.79</td>
<td>xrelay</td>
<td>8.36411</td>
</tr>
<tr>
<td>PEAR_S_POST_HZ</td>
<td>pXRelay_PEAR_S</td>
<td>44.74</td>
<td>xrelay</td>
<td>8.36406</td>
</tr>
</tbody>
</table>

Upon each write to the MOOSDB the value of the variable is incremented by 1, and the integer progression can be monitored in the last column on lines 2-3. The APPLES\_POST\_HZ and PEAR\_S\_POST\_HZ variables represent the frequency at which the process makes a post to the MOOSDB. This of course is different than (but bounded above by) the frequency of the Iterate() loop since a post is made within the Iterate() loop only if mail had been received prior to the outset of the loop. In a world with no latency, one might expect the "post" frequency to be exactly half of the "iterate" frequency.
We would expect the frequency reported on lines 6-7 to be no greater than 12.5, and in this case values of about 8.4 are observed instead.

### 3.8.4 The pXRelay Example MOOS Configuration File

The mission file used for the pXRelay example, xrelay.moos is discussed here. This file is provided as part of the MOOS-IvP software bundle under the “missions” directory as discussed above in Section 3.8.1. It is discussed here in three parts in Listings 4 through 6 below.

The part of the xrelay.moos file provides three mandatory pieces of information needed by the MOOSDB process for launching. The MOOSDB is a server and on line 1 is the IP address for the machine, and line 2 indicates the port number where clients can expect to find the MOOSDB once it has been launched. Since each MOOSDB and the set of connected clients form a MOOS “community”, the community name is provided on line 3. Note the xrelay community name in the xrelay.moos file and the community name in column 4 of the uXMS output in Listing 1 above.

Listing 3.4: The xrelay.moos mission file for the pXRelay example.

```moos
1 ServerHost = localhost
2 ServerPort = 9000
3 Community = xrelay
4
5 //------------------------------------------
6 // Antler configuration block
7 ProcessConfig = ANTLER
8 {
9   MSBetweenLaunches = 200
10  
11  Run = MOOSDB @ NewConsole = true
12  Run = pXRelay @ NewConsole = true ~ pXRelay_PEARS
13  Run = pXRelay @ NewConsole = true ~ pXRelay_APPLES
14  Run = uXMS @ NewConsole = true
15 }
```

The configuration block in lines 7-15 of xrelay.moos is read by the pAntler for launching the processes or clients of the MOOS community. Line 9 specifies how much time, in milliseconds, between the launching of processes. Lines 11-14 name the four MOOS applications launched in this example. On these lines, the component "NewConsole = true" determines whether a new console window will be opened for each process. Try changing them to false - only the uXMS window really needs to be open. The others merely provide a visual confirmation that a process has been launched. The " ~ pXRelay_PEARS" component of lines 12 and 13 tell pAntler to launch these applications with the given alias. This is required here since each MOOS client needs to have a unique name, and in this example two instances of the pXRelay process are being launched.

In lines 17-39 in Listing 5-B below, the two pXRelay applications are configured. Note that the argument to ProcessConfig on lines 20 and 32 is the alias for pXRelay specified in the Antler configuration block on lines 12 and 13. Each pXRelay process is configured such that its incoming and outgoing MOOS variables complement one another on lines 25-26 and 37-38. Note the AppTick parameter (see Section 3.4.1) is set to 25 in both configuration blocks, and compare with the observed frequency of the Iterate() function reported in the variables APPLES_ITER_HZ and PEARS_ITER_HZ in Listing 1. MOOS has done a pretty faithful job in this example of honoring the requested frequency of the Iterate() loop in each application.

Listing 3.5: The xrelay.moos mission file - configuring the pXRelay processes.
In the last portion of the xrelay.moos file, shown in Listing 6-C below, the uXMS process is configured. In this example, uXMS is configured to scope on the six variables specified on lines 54-59 to give the output shown in Listings 1 and 3. By setting the paused parameter on line 49 to false, the output of uXMS is continuously and automatically updated - in this case four times per second due to the rate of 4Hz specified in lines 46-47. The display.* parameters in lines 50-52 ensure that the output in columns 2-4 of the uXMS output is expanded. See [31] for further ways to configure the uXMS tool.

Listing 3.6: Configuring uXMS in the pXRelay example.

3.8.5 Suggestions for Further Things to Try with this Example

- Take a look at the OnStartUp() method in the XRelay.cpp class in the pXRelay module in the software bundle to see how the handling of parameters in the xrelay.moos configuration file
are implemented, and the subscription for a MOOS variable.

- Take a look at the `OnNewMail()` method in the `XRelay.cpp` class in the `pXRelay` module in the software bundle to see how incoming mail is parsed and handled.

- Take a look at the `Iterate()` method in the `XRelay.cpp` class in the `pXRelay` module in the software bundle to see an example of a MOOS process that acts upon incoming mail and conditionally posts to the MOOSDB.

- Try changing the `AppTick` parameter in one of the `pXRelay` configuration blocks in the `xrelay.moos` file, re-start, and note the resulting change in the iteration and post frequencies in the `uXMS` output.

- Try changing the `CommsTick` parameter in one of the `pXRelay` configuration blocks in the `xrelay.moos` file to something much lower than the `AppTick` parameter, re-start, and note the resulting change in the iteration and post frequencies in the `uXMS` output.

### 3.9 MOOS Applications Available to the Public

Below are very brief descriptions of MOOS applications in the public domain. This is by no means a complete list. It does not include applications outside MIT and Oxford, and it is not even a complete list of applications from those organizations. For a more in-depth tour of MOOS applications, see [31].

#### 3.9.1 MOOS Modules from Oxford

- **pAntler**: A tool for launching a collection of MOOS processes given a mission file. See [47], [46].
- **pShare**: A tool that allows messages to pass between communities and allows for the renaming of messages as they are shuffled between communities. See [46].
- **pLogger**: A logger for recording the activities of a MOOS session. It can be configured to record a fraction of, or all publications of any number of MOOS variables. See [46].
- **pScheduler**: A simple tool for generating and responding to messages sent to the MOOSDB by processes in a MOOS community. See [46].
- **uMS**: A GUI-Based MOOS scope for monitoring one or more MOOSDBs. See [46].
- **uPlayback**: An FLTK-based, cross platform GUI application that can load in log files and replay them into a MOOS community as though the originators of the data were really running and issuing notifications. See [46].
- **iMatlab**: An application that allows Matlab to join a MOOS community - even if only for listening in and rendering sensor data. It allows connection to the MOOSDB and access to local serial ports. See [46].
- **iRemote**: A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associate a pre-defined variable-value poke with any un-mapped key on the keyboard. See [46].
- **uMVS**: A multi-vehicle AUV simulator, capable of simulating any number of vehicles and acoustic ranging between them and acoustic transponders. The vehicle simulation incorporates a full 6 D.O.F vehicle model replete with vehicle dynamics, center of buoyancy / center of gravity.
gravity geometry, and velocity dependent drag. The acoustic simulation is also fairly smart. It simulates acoustic packets propagating as spherical shells through the water column. See [46].

3.9.2 Mission Monitoring Modules

Mission monitoring modules aid the user in either keeping a high-level tab on the mission as it unfolds, or help the user analyze and debug a mission. In release 13.2 this includes two powerful new tools for appcast monitoring, uMAC and uMACView. The pMarineViewer has also been substantially augmented to support appcast viewing.

- **pMarineViewer**: GUI tool for rendering events in an area of vehicle operation. It repeatedly updates vehicle positions from incoming node reports, and will render several geometric types published from other MOOS apps. The viewer may also post messages to the MOOSDB based on user-configured keyboard or mouse events. See the online documentation for pMarineViewer, [5].

- **uHelmScope**: A terminal-based (non-GUI) scope onto a running IvP Helm process, and key MOOS variables. It provides behavior summaries, activity states, and recent behavior postings to the MOOSDB. A very useful tool for debugging helm anomalies. See the documentation for uHelmScope, [22].

- **uXMS**: A terminal-based (non-GUI) tool for scoping a MOOSDB. Users may precisely configure the set of variables they wish to scope on by naming them explicitly on the command line or in the MOOS configuration block. The variable set may also be configured by naming one or more MOOS processes on which all variables published by those processes will be scoped. Users may also scope on the history of a single variable. See the documentation for uXMS, [28].

- **uProcessWatch**: This application monitors the presence of MOOS apps on a watch-list. If one or more are noted to be absent, it will be so noted on the MOOS variable PROC_WATCH_SUMMARY. uProcessWatch is appcast-enabled and will produce a succinct table summary of watched processes and the CPU load reported by the processes themselves. The items on the watch list may be named explicitly in the config file or inferred from the Antler block or from list of DB_CLIENTS. An application may be excluded from the watch list if desired. See the documentation for uProcessWatch, [24].

- **uMAC**: The uMAC application is a utility for Monitoring AppCasts. It is launched and run in a terminal window and will parse appcasts generated within its own MOOS community or those from other MOOS communities bridged or shared to the local MOOSDB. The primary advantage of uMAC versus other appcast monitoring tools is that a user can remotely log into a vehicle via ssh and launch uMAC locally in a terminal. See the documentation for uMAC, [13].

- **uMACView**: A GUI tool for visually monitoring appcasts. It will parse appcasts generated within its own MOOS community or those from other MOOS communities bridged or shared to the local MOOSDB. Its capability is nearly identical to the appcast viewing capability built into pMarineViewer. It was intended to be an appcast viewer for non-pMarineViewer users. See the documentation for uMACView, [14].

3.9.3 Mission Execution Modules

Mission execution modules participate directly in the proper execution of the mission rather than simply helping to monitor, plan or analyze the mission.
• **pNodeReporter**: A tool for collecting node information such as present vehicle position, trajectory and type, and posting it in a single report for sharing between vehicles or sending to a shoreside display. See the documentation for pNodeReporter, [6].

• **pBasicContactMgr**: The contact manager deals with other known vehicles in its vicinity. It handles incoming reports perhaps received via a sensor application or over a communications link. Minimally it posts summary reports to the MOOSDB, but may also be configured to post alerts with user-configured content about one or more of the contacts. May be used in conjunction with the helm to spawn contact-related behaviors for collision avoidance, tracking, etc. See the documentation for pBasicContactMgr, [3].

• **pEchoVar**: A tool for subscribing for a variable and re-publishing it under a different name. It also may be used to pull out certain fields in string publications consisting of comma-separated parameter=value pairs, publishing the new string using different parameters. See the documentation for pEchoVar, [4].

• **pSearchGrid**: An application for storing a history of vehicle positions in a 2D grid defined over a region of operation.

3.9.4 Mission Simulation Modules

Mission simulation modules are used only in simulation. Many of the applications in the uField Toolbox may also be considered simulation modules, but they also have a use case involving simulated sensors on actual physical vehicles. The two modules below are purely for simulated vehicles.

• **uSimMarine**: A simple 3D vehicle simulator that updates vehicle state, position and trajectory, based on the present actuator values and prior vehicle state. Typical usage scenario has a single instance of uSimMarine associated with each simulated vehicle. See the documentation for uSimMarine, [25].

• **uSimCurrent**: A simple application for simulating the effects of water current. Based on local current information from a given file, it repeatedly reads the vehicle’s present position and publishes a drift vector, presumably consumed by uSimMarine.

3.9.5 Modules for Poking the MOOSDB

Poking the MOOSDB is a common and often essential part of mission execution and/or command and control. The pMarineViewer tool also contains several methods for poking the MOOSDB on user command.

• **uPokeDB**: A command-line tool for poking a MOOSDB with variable-value pairs provided on the command line. It finds the MOOSDB via mission file provided on the command line, or the IP address and port number given on the command line. It will connect to the DB, show the value prior to poking, poke the DB, and wait for mail from the DB to confirm the result of the poke. See the documentation for uPokeDB, [23].

• **uTimerScript**: Allows the user to script a set of pre-configured pokes to a MOOSDB with each entry in the script happening after a specified amount of time. Script may be paused or fast-forwarded. Events may also be configured with random values and happen randomly in a chosen window of time. See the documentation for uTimerScript, [27].
• **uTermCommand**: A terminal application for poking the MOOSDB with pre-defined variable-value pairs. A unique key may be associated with each poke. See the documentation for *uTermCommand*, [26].

### 3.9.6 The Alog Toolbox

The Alog Toolbox is set of offline tools for analyzing and manipulating alog files produced by the *pLogger* application distributed with the Oxford MOOS codebase.

- **alogscan**: A command line tool for reporting the contents of a given MOOS .alog file. See the documentation for *alogscan*, part of the Alog Toolbox, [11].
- **alogclip**: A command line tool that will create a new MOOS .alog file from a given .alog file by removing entries outside a given time window. See the documentation for *alogclip*, part of the Alog Toolbox, [8].
- **aloggrep**: A command line tool that will create a new MOOS .alog file by retaining only the given MOOS variables or sources from a given .alog file. See the documentation for *aloggrep*, part of the Alog Toolbox, [9].
- **alogrm**: A command line tool that will create a new MOOS .alog file by removing the given MOOS variables or sources from a given .alog file. See the documentation for *alogrm*, part of the Alog Toolbox, [10].
- **alogview**: A GUI tool for analyzing a vehicle mission by plotting one or more vehicle trajectories on the operation area, while viewing a plot of any of the numerical values in the alog file(s). See the documentation for *alogview*, part of the Alog Toolbox, [12].

### 3.9.7 The uField Toolbox

The uField Toolbox contains a number of tools for supporting multi-vehicle missions where each vehicle is connected to a shoreside community. This includes both simulation and real field experiments. It also contains a number of simulated sensors that run on offboard the vehicle on the shoreside.

- **pHostInfo**: Automatically detect the vehicle’s host information including the IP addresses, port being used by the MOOSDB, the port being used by local pShare for UDP listening, and the community name for the local MOOSDB. Post these to facilitate automatic intervehicle communications in especially in multi-vehicle scenarios where the local IP address changes with DHCP.
- **uFldNodeBroker**: Typically run on a vehicle or simulated vehicle in a multi-vehicle context. Used for making a connection to a shoreside community by sending local information about the vehicle such as the IP address, community name, and port number being used by pShare for incoming UDP messages. Presumably the shoreside community uses this to know where to send outgoing UDP messages to the vehicle. See the documentation for *uFldNodeBroker*, part of the uField Toolbox, [18].
- **uFldShoreBroker**: Typically run in a shoreside community. Takes reports from remote vehicles describing how they may be reached. Posts registration requests to shoreside pShare to bridge user-provided list of variables out to vehicles. Upon learning of vehicle JAKE will
create bridges FOO_ALL and FOO_JAKE to JAKE, for all such user-configured variables. See the documentation for uFldShoreBroker, part of the uField Toolbox, [21].

- **uFldNodeComms**: A shoreside tool for managing communications between vehicles. It has knowledge of all vehicle positions based on incoming node reports. Communications may be limited based on vehicle range, frequency of messages, or size of message. Messages may also be blocked based on a team affiliation. See the documentation for uFldNodeComms, part of the uField Toolbox, [19].

- **uFldMessageHandler**: A tool for handling incoming messages from other nodes. The message is a string that contains the source and destination of the message as well as the MOOS variable and value. This app simply posts to the local MOOSDB the variable-value pair contents of the message. See the documentation for uFldMessageHandler, part of the uField Toolbox, [17].

- **uFldScope**: Typically run in a shoreside community. Takes information from user-configured set of incoming reports and parses out key information into a concise table format. Reports may be any report in the form of comma-separated parameter-value pairs.

- **uFldPathCheck**: Typically run in a shoreside community. Takes node reports from remote vehicles and calculates the current vehicle speed and total distance travelled and posts them in two concise reports. Odometry tallies may be re-set to zero by other apps. See the documentation for uFldPathCheck, part of the uField Toolbox, [20].

- **uFldHazardSensor**: Typically run in a shoreside community. Configured with a set objects with a given x,y location and classification (hazard or benign). The sensor simulator receives a series of requests from a remote vehicle. When sensor determines that an object is in the sensor field of a requesting vehicle, it may or may not return a sensor detection report for the object, and perhaps also a proper classification. The odds of receiving a detection and proper classification depend on the sensor configuration and the user’s preference for $P_D/P_{FA}$ on the prevailing ROC curve.

- **uFldHazardMetric**: An application for grading incoming hazard reports, presumably generated by users of the uFldHazardSensor after exploring a simulated hazard field.

- **uFldHazardMgr**: The uFldHazardMgr is a strawman MOOS app for managing hazard sensor information and generation of a hazard report over the course of an autonomous search mission.

- **uFldBeaconRangeSensor**: Typically run in a shoreside community. Configured with one or more beacons with known beacon locations. Takes range requests from a remote vehicle and returns a range report indicating that vehicle’s range to nearby beacons. Range requests may or may not be answered depending on range to beacon. Reports may have noise added and may or may not include beacon ID. See the documentation for uFldBeaconRangeSensor, part of the uField Toolbox, [15].

- **uFldContactRangeSensor**: Typically run in a shoreside community. Takes reports from remote vehicles, notes their position. Takes a range request from a remote vehicle and returns a range report indicating that vehicle’s range to nearby vehicles. Range requests may or may not be answered dependent on inter-vehicle range. Reports may also have noise added to their range values. See the documentation for uFldContactRangeSensor, part of the uField Toolbox, [16].
4 A First Example with MOOS-IvP - the Alpha Mission

This section describes a simple mission using the helm. It is designed to run in simulation on a single machine. The mission configuration files for this example are distributed with the source code. Information on how to find these files and launch this mission are described below in Section 4.1. In this example the vehicle simply traverses a set of pre-defined given waypoints and returns back to the launch position. The user may return the vehicle any time before completing the waypoints, and may subsequently command the vehicle to resume the waypoints at any time. This example touches on the following issues:

- Launching a mission with a given mission (.moos) file and behavior (.bhv) file.
- Configuration of MOOS processes, including the IvP Helm, with a .moos file.
- Configuration of the IvP Helm (mission planning) with a .bhv file.
- Implementation of simple command and control with the IvP Helm.
- Interaction between MOOS processes and the helm during normal mission operation.

Here is a bit of what the Alpha mission should look like:

![Figure 6: The Alpha mission.](https://vimeo.com/84549446)

4.1 Find and Launch the Alpha Example Mission

The example mission should be in the same directory tree containing the source code. There are two files - a MOOS file, also mission file or .moos file, and a behavior file or .bhv file:
To run this mission from a terminal window, simply change directories and launch:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ pAntler alpha.moos
```

After pAntler has launched each process, the pMarineViewer window should be open and look similar to that shown in Figure 7. After clicking the DEPLOY button in the lower right corner the vehicle should start to traverse the shown set of waypoints.

![Figure 7: The Alpha Example Mission - In the Surveying Mode](image)

This mission will complete on its own with the vehicle returning to the launch point. Alternatively,
by hitting the RETURN button at any time before the points have been traverse, the vehicle will change course immediately to return to the launch point, as shown in Figure 8. When the vehicle is returning as in the figure, it can be re-deployed by hitting the DEPLOY button again.

Figure 8: The Alpha Example Mission - In the Returning Mode: The vehicle can be commanded to return prior to the completion of its waypoints by the user clicking the RETURN button on the viewer.

The vehicle in this example is configured with two basic waypoint behaviors. Their configuration with respect to the points traversed and when each behavior is actively influencing the vehicle, is discussed next.

4.2 A Look at the Behavior File used in the Alpha Example Mission

The mission configuration of the helm behaviors is provided in a behavior file, and the complete behavior file for the example mission is shown in Listing 1. Behaviors are configured in blocks of parameter-value pairs - for example lines 6-17 configure the waypoint behavior with the five waypoints shown in the previous two figures. This is discussed in more detail in Section 6.3.

Listing 4.1: The behavior file for the Alpha example.

```
1 initialize DEPLOY = false
2 initialize RETURN = false
```
Behavior = BHV_Waypoint
{
  name = waypt_survey
  pwt = 100
  condition = RETURN = false
  condition = DEPLOY = true
  endflag = RETURN = true
  perpetual = true
  lead = 8
  lead_damper = 1
  speed = 2.0 // meters per second
  radius = 4.0
  nm_radius = 10.0
  points = 60, -40:60, -160:150, -160:180, -100:150, -40
  repeat = 1
}

Behavior = BHV_Waypoint
{
  name = waypt_return
  pwt = 100
  condition = RETURN = true
  condition = DEPLOY = true
  perpetual = true
  endflag = RETURN = false
  endflag = DEPLOY = false
  speed = 2.0
  radius = 2.0
  nm_radius = 8.0
  point = 0,0
}

The parameters for each behavior are separated into two groups. Parameters such as name, priority, condition and endflag are parameters defined generally for all IvP behaviors. Parameters such as speed, radius, and points are defined specifically for the Waypoint behavior. A convention used in .bhv files is to group the general behavior parameters separately at the top of the configuration block.

In this mission, the vehicle follows two sets of waypoints in succession by configuring two instances of a basic waypoint behavior. The second waypoint behavior (lines 23-37) contains only a single waypoint representing the vehicle launch point (0,0). It’s often convenient to have the vehicle return home when the mission is completed - in this case when the first waypoint behavior has reached its last waypoint. Although it’s possible to simply add (0,0) as the last waypoint of the first waypoint behavior, it is useful to keep it separate to facilitate recalling the vehicle pre-maturely at any point after deployment.

Behavior conditions (lines 9-10, 28-29), and endflags (line 110, lines 31-32) are primary tools for coordinating separate behaviors into a particular mission. Behaviors will not participate unless each of its conditions are met. The conditions are based on current values of the MOOS variables involved in the condition. For example, both behaviors will remain idle unless the variable DEPLOY is set to true. This variable is set initially to be false by the initialization on line 1, and is toggled by the DEPLOY button on the pMarineViewer GUI shown in Figures 7 and 8. The pMarineViewer MOOS application is one option for a command and control interface to the helm. The MOOS variables in
the behavior conditions in Listing 1 do not care which process was responsible for setting the value. Endflags are used by behaviors to post a MOOS variable and value when a behavior has reached a completion. The notion of completion is different for each behavior and some behaviors have no notion of completion, but in the case of the waypoint behavior, completion is declared when the last waypoint is reached. In this way, behaviors can be configured to run in a sequence, as in this example, where the returning waypoint behavior will have a necessary condition (line 28) met when the surveying behavior posts its endflag on line 11.

4.3 A Closer Look at the MOOS Apps in the Alpha Example Mission

Running the example mission involves five other MOOS applications in addition to the IvP helm. In this section we take a closer look at what those applications do and how they are configured. The full MOOS file, alpha.moos, used to run this mission is given in full in the appendix. An overview of the situation is shown in Figure 9.

![The MOOS processes in the example "alpha" mission: In (1) The helm produces a desired heading and speed. In (2) the PID controller subscribes for the desired heading and speed and publishes actuation values. In (3) the simulator grabs the actuator values and the current vehicle pose and publishes a set of MOOS variables representing the new vehicle pose. In (4) all navigation output is wrapped into a single node-report string to be consumed by the helm and the GUI viewer. In (5) the pMarineViewer grabs the node-report and renders a new vehicle position. The user can interact with the viewer to write limited command and control variables to the MOOSDB.](image)

4.3.1 Antler and the Antler Configuration Block

The pAntler tool is used to orchestrate the launching of all the MOOS processes participating in this example. From the command line, pAntler is run with a single argument the .moos file. As it launches processes, it hands each process a pointer to this same MOOS file. The Antler configuration block in this example looks like:

Listing 4.2: An example Antler configuration block for the Alpha mission.
ProcessConfig = ANTLER
{
  MSBetweenLaunches = 200

  Run = MOOSDB @ NewConsole = false
  Run = uSimMarine @ NewConsole = false
  Run = pNodeReporter @ NewConsole = false
  Run = pMarinePID @ NewConsole = false
  Run = pMarineViewer @ NewConsole = false
  Run = pHelmIvP @ NewConsole = false
}

The first parameter on line 2 specifies how much time should be left between the launching of each process. Lines 4-9 specify which processes to launch. The MOOSDB is typically launched first. The NewConsole switch on each line determines whether a new console window should be opened with each process. Try switching one or more of these to true as an experiment.

4.3.2 The pMarinePID Application

The pMarinePID application implements a simple PID controller which produces values suitable for actuator control based on inputs from the helm. In simulation the output is consumed by the vehicle simulator rather than the vehicle actuators.

In short: The pMarinePID application typically gets its info from pHelmIvP; produces info consumed by uSimMarine or actuator MOOS processes when not running in simulation.

Subscribes to: DESIRED_HEADING, DESIRED_SPEED.
Publishes to: DESIRED_RUDDER, DESIRED_THRUST.

4.3.3 The uSimMarine Application and Configuration Block

The uSimMarine application is a very simple vehicle simulator that considers the current vehicle pose and actuator commands and produces a new vehicle pose. It can be initialized with a given pose as shown in the configuration block used in this example, shown in Listing 3:

Listing 4.3: An example uSimMarine configuration block for the Alpha mission.

  ProcessConfig = uSimMarine
  {
    AppTick = 10
    CommsTick = 10

    START_X = 0
    START_Y = 0
    START_SPEED = 0
    START_HEADING = 180
    PREFIX = NAV
  }

In short: The uSimMarine application typically gets its info from pMarinePID; produces info consumed by pNodeReporter and itself on the next iteration of uSimMarine.
Subscribes to: DESIRED_RUDDER, DESIRED_THRUST, NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING.

Publishes to: NAV_X, NAV_Y, NAV_HEADING, NAV_SPEED.

4.3.4 The pNodeReporter Application and Configuration Block

An Automated Information System (AIS) is commonplace on many larger marine vessels and is comprised of a transponder and receiver that broadcasts one's own vehicle ID and pose to other nearby vessels equipped with an AIS receiver. It periodically collects all latest pose elements, e.g., latitude and longitude position and latest measured heading and speed, and wraps it up into a single update to be broadcast. This MOOS process collects pose information by subscribing to the MOOSDB for NAV_X, NAV_Y, NAV_HEADING, NAV_SPEED, and NAV DEPTH and wraps it up into a single MOOS variable called NODE_REPORT_LOCAL. This variable in turn can be subscribed to another MOOS process connected to an actual serial device acting as an AIS transponder. For our purposes, this variable is also subscribed to by pMarineViewer for rendering a vehicle pose sequence.

In short: The pNodeReporter application typically gets its info from uSimMarine or otherwise onboard navigation systems such as GPS or compass; produces info consumed by pMarineViewer and instances of pHelmIVP running in other vehicles or simulated vehicles.

Subscribes to: NAV_X, NAV_Y, NAV_SPEED, NAV_HEADING.

Publishes to: NODE_REPORT_LOCAL

4.3.5 The pMarineViewer Application and Configuration Block

The pMarineViewer is a MOOS process that subscribes to the MOOS variable NODE_REPORT_LOCAL and NODE_REPORT which contains a vehicle ID, pose and timestamp. It renders the updated vehicle(s) position. It is a multi-threaded process to allow both communication with MOOS and let the user pan and zoom and otherwise interact with the GUI. It is non-essential for vehicle operation, but essential for visually confirming that all is going as planned.

In short: The pMarineViewer application typically gets its info from pNodeReporter and pHelmIVP; produces info consumed by pHelmIVP when configured to have command and control hooks (as in this example).

Subscribes to: NODE_REPORT, NODE_REPORT_LOCAL, VIEW_POINT, VIEW_SEGLIST, VIEW_POLYGON, VIEW_MARKER.

Publishes to: Depends on configuration, but in this example: DEPLOY, RETURN.
5 The IvP Helm as a MOOS Application

In this section the helm is discussed in terms of its identity as a MOOS application - its MOOS configuration parameters, its `Iterate()` loop, its output to the console, and its output in terms of publications to other applications running in the MOOS community. The `helm state` and `all-stop status` are also introduced since they are the highest level descriptions regarding helm activity.

5.1 Overview

The IvP Helm is implemented as the MOOS module called `pHelmIvP`. On the surface it is similar to any other MOOS application - it runs as a single process that connects to a running MOOSDB process interfacing solely by a publish-subscribe interface, as depicted in Figure 10. It is configured from a behavior file, or `.bhv` file, in addition to the MOOS file used to configure other MOOS applications. The helm primarily publishes a steady stream of information that drives the platform, typically regarding the desired heading, speed or depth. It may also publish information conveying aspects of the autonomy state that may be useful for monitoring, debugging or triggering other algorithms either within the helm or in other MOOS processes. The helm can be configured to generate decisions over virtually any user-defined decision space.

![Figure 10: The pHelmIvP MOOS application](image)

The helm subscribes for sensor information or any other information it needs to make decisions. This information includes navigation information regarding the platform’s current position and trajectory, information regarding the position or state of other vehicles, or environmental information. The information it subscribes for is prescribed by the behaviors themselves, configured in the `.bhv`
file. In addition to sensor information, the helm also receives some level of command and control information. For example, in some marine vehicle configurations, one of the "Other MOOSApp" modules in the figure is a driver for an acoustic modem over which command and control information may be relayed.

The helm has a couple informative high-level state descriptions, the helm state and the all-stop status, that may be compared to the operation of an automobile. Launching the pHelmIVP MOOS process is analogous to turning on the car’s engine. Putting the helm in the DRIVE mode is like shifting the car from "Park" to "Drive". And the all-stop status refers whether or not the car is breaking to a stop. The analogy is summarized below.

5.2 The Helm State

The highest level interface with the helm is reflected in the helm state. The helm state may have one of two values, park or drive (except when using a standby helm described in Section 5.7) In the drive mode, the helm is likely in the process of executing a mission. In the park mode, the helm is waiting, likely because it is being asked to wait, but also because the helm may have noticed something wrong (generated an all-stop) and subsequently put itself into park on its own, awaiting the chance to return to the drive state. In this section we discuss (a) how the helm state is changed, (b) what it is going on in the helm when it is parked, and (c) how the helm state is initialized at start-up. At any point in time after the helm is launched, the helm will post the MOOS variable IVPHELM_STATE with either the value "DRIVE", "PARK" or "MALCONFIG". This is posted on each iteration and registering for this mail is the manner recommended by which other MOOS applications monitor the helm’s heart beat.

5.2.1 Helm State Transitions

The helm state may be transitioned by writing to the MOOS variable MOOS_MANUAL_OVERRIDE. As Figure 12 depicts, a value of false, which is case insensitive, transitions the helm state into DRIVE. A value of true puts it into the PARK. When the helm transitions from DRIVE to PARK it makes one more publication to the helm decision variables, each with a value zero. This is referred to as the
production of an all-stop posting, discussed in more detail in a later section.

Figure 12: The Helm State of the IvP Helm: The helm state has a value of either PARK or DRIVE, depending on both how the helm is initialized and the mail received by the helm after start-up on the variable MOOS_MANUAL_OVERRIDE. The helm may also park itself if an all-stop event has been detected.

The variable MOOS_MANUAL_OVERRIDE contains the mis-spelling of ”override”. However, it is a variable that has some legacy presence in other MOOS applications such as iRemote. To avoid a situation where there is an attempt to override the helm, but the request is ignored because of a (proper) spelling, the helm will also respect transition requests on the properly spelled variable MOOS_MANUAL_OVERRIDE. This has the drawback however that these two variables could conceivably have different values in the MOOSDB. This is not a problem but could be confusing for someone trying to infer the helm state by opening a scope on the MOOSDB, on either the wrong variable or the two disagreeing variables. In this case the helm state would be aligned with the variable with the most recent publication time stamp. In any event, the best way to monitor the helm state is to scope on the MOOS variable IVPHELM_STATE, published by the helm itself, or use the uHelmScope tool.

The helm may also automatically transition itself from the DRIVE to the PARK state (but never the other way around), by posting an all-stop event. All-stop events and the helm are discussed separately in Section 5.3. All-stop events are generated by the helm upon finding that one or more possible error conditions have been detected during the normal execution of the helm iteration. If the helm parks due to an all-stop, it may be returned to the DRIVE state by another MOOS client posting MOOS_MANUAL_OVERRIDE=false, but this is no guarantee that the helm wouldn’t just park again immediately if the same condition persists that caused the all-stop event.

5.2.2 What Is and Isn’t Happening when the Helm is Parked

When the helm state is PARK, the MOOS application loop depicted in Figure 5 carries on. The OnNewMail() continues to be called and new mail is read and dealt with exactly as it would if the helm state were DRIVE. The Iterate() loop, however, is truncated to virtually a no-op, with the only action being the output of a heartbeat character to the console if the helm is configured to do so (Section 5.5). No behavior code is called whatsoever. The helm iteration counter, a key index in the uHelmScope output, is also suspended despite the fact that technically the Iterate() loop
continues to be called.

5.2.3 Initializing the Helm State at Process Launch Time

The helm, by default, is configured to be initially in PARK upon start-up. By setting the parameter `start_in_drive=true` in the mission file configuration block, the helm will indeed be in the DRIVE upon start-up. This feature was found to have practical use in UUV operations to allow for rebooting of the autonomy computer to automatically launch the helm, in DRIVE and ready to accept field commands. This feature should be used with caution, and it may be phased out in a later software release.

5.3 Helm All-Stop Events and All-Stop Status

An all-stop event is something that brings the vehicle to a full-stop, with typically zero-speed, zero-depth commanded. The all-stop status is simply a string describing why the vehicle is at an all-stop. Sometimes an all-stop indicates a problem, e.g., missing critical sensor information. Sometimes a vehicle may just stop as part of the mission, e.g., coming to the surface for a GPS fix. The all-stop status message can be used to discern the two types of situations. In the automobile analogy, an all-stop is equivalent to hitting the car’s breaks with the intent to stop completely. An all-stop event will result in the following:

- Zero-values will be posted for all decision variables, `DESIRED_SPEED=0`, `DESIRED_DEPTH=0`, etc.
- The helm will possibly transition into PARK. By default the helm is configured to remain in the DRIVE upon an all-stop event, but if configured instead with `park_on_allstop = true`, the helm will indeed park upon an all-stop.
- The reason for the all-stop will be posted to MOOS variable `IVPHELM_ALLSTOP`. The value of this variable will be "clear" if there are no all-stop events that have occurred since the helm has entered the DRIVE state.

The reasons for all-stop may be:

- No behaviors are active. The helm has absolutely no opinion about any of its decision variables. In this case, the following would be posted: `IVPHELM_ALLSTOP ="NothingToDo"`.  
- Some behaviors are active, but decisions are missing on one or more mandatory decision variables. In this case, the following would be posted: `IVPHELM_ALLSTOP ="MissingDecVars"`.  
- When the vehicle is parked due to manual override, the following would be posted: `IVPHELM_ALLSTOP ="ManualOverride"`.  
- One of the behaviors has determined an all-stop is warranted for some reason. For example, a waypoint behavior that cannot determine own-platform’s current position would declare an all-stop. In this case, the following would be posted: `IVPHELM_ALLSTOP ="BehaviorError"`.  

To gain further insight on an all-stop caused by a behavior error, the nature of the error is expressed in a separate posting to the MOOS `BHV_ERROR` variable. It’s possible that more than one behavior
error occurred in the helm iteration where the all-stop event occurred, in which case there would be multiple postings to the `BHV_ERROR` variable. When the vehicle is in `DRIVE` and operating free of an all-stop, the all-stop status is reflected by `IVPHELM_ALLSTOP = "clear"`.

5.4 Parameters for the pHelmIvP MOOS Configuration Block

The following configuration parameters are defined for the helm. The parameter names are case insensitive.

Listing 5.1: Configuration Parameters for pHelmIvP.

- **allow_park**: If false, the helm cannot be put into `PARK`. This parameter is not mandatory. The default is true. Section 5.4.1.
- **behaviors**: The name and location of the behavior configuration file. This parameter is not mandatory, but typically it is used. Technically the helm can be launched from the command line and provided the behavior file on the command line. Section 5.4.2.
- **ivp_behavior_dir**: A directory to look for dynamically loaded behaviors. This parameter is not mandatory, since the directory information may also be handled using a shell environment variable. Section 5.4.3.
- **community**: Global MOOS parameter. Determines ownship name. This parameter is mandatory, but it is provided outside the helm configuration block and used by other applications. Section 5.4.4.
- **park_on_allstop**: If true helm will park on all-stop. This parameter is not mandatory and the default is false. Section 5.4.1.
- **domain**: The decision space for the IvP Solver. This parameter is mandatory. Section 5.4.6.
- **hold_on_app**: A list of MOOS apps to wait for before the helm publishes postings from behaviors’ `onHelmStart()` function calls. Available after Release 17.7.x. Section 5.4.10.
- **ok_skew**: The tolerance on the age, in seconds, of incoming mail before rejected as being too old. This parameter is not mandatory. Section 5.4.7.
- **other_override_var**: The parameter names an additional MOOS variable acting as `MOOS_MANUAL_OVERRIDE`. This parameter is not mandatory. Section 5.4.8.
- **start_in_drive**: Determines whether or not the helm is in override mode at start-up. This parameter is not mandatory. The default is false. Section 5.4.9.
- **helm_prefix**: Add a prefix to all the `DESIRED_*` helm output. For example `helm_prefix=FOO` would result in `FOO_DESIRED_SPEED` and so on. Introduced after Release 19.8.x
- **verbose**: Determines verbosity of terminal output - `quiet`, `terse`, or `verbose`. This parameter is not mandatory. The default is `verbose`. Section 5.4.11.
5.4.1 The allow_park Parameter

Optional. By setting this parameter to false, the helm cannot be manually overridden (parked) once it has been put into DRIVE. This can be dangerous and should be carefully considered, and thus the default is true. This option was implemented based on experiences with launching UUV autonomy missions and preventing an inadvertent park due to a remote login to the vehicle. There was a tendency for some users to use iRemote upon remote login to interact with MOOS, and iRemote posts MOOS_MANUAL_OVERRIDE =true upon launch and connection to the MOOSDB.

5.4.2 The behaviors Parameter

Optional (sort of). The parameter names the behavior file, i.e., *.bhv file, on the local file system from which the helm behaviors are read. More than one file may be specified on separate lines, and the helm will read in all files almost as if they were one single file. This is technically an optional parameter because a behavior file could be provide on the command line. A behavior file must be specified via one means or the other. If a behavior file is specified both on the command line and in the pHelmIvP configuration block with this parameter, they will both be used to configure the helm behaviors.

5.4.3 The behavior_dir Parameter

Optional. The parameter names a directory in the local file system where the helm is to look for dynamically loadable behaviors (as opposed to default set built in statically to the helm). Authors augmenting the helm with their own behaviors will need to specify the location of those behaviors with this parameter. More than one line may be provided, each specifying a different directory location.

5.4.4 The community Parameter

This parameter is defined at the "global" level outside of any MOOS process’ configuration block. See Section 3.5. The helm reads this parameter and uses its value as the name associated with "ownship". It is a mandatory parameter.

5.4.5 The park_on_allstop Parameter

Optional. By setting this parameter to true, the helm will park when an all-stop event occurs. The default setting is false.

5.4.6 The domain Parameter

Mandatory. This parameter prescribes the decision space of the helm. It consists of one line per decision variable. Each line contains a colon-separated list of four fields. Field one is the domain variable name, field two is the lower bound value, field three is the higher bound value, and field four is the number of points in the domain. For example domain = speed:0:3:16 shown in Listing 2 indicates a domain variable called "speed", with a lower and upper bound 0 and 3 meters/second respectively. Since there are 16 points, the speed choices are 0, 0.2, 0.4, ..., 2.8, 3.0. The helm requires that a decision be made on all listed variables on each iteration of the control loop. If a
variable is used by some behaviors but is not necessarily involved in all decisions, it can be declared as optional. For example `domain=speed:0:3:16:optional`.

5.4.7 The `ok_skew` Parameter

Optional. This parameter sets the allowable skew tolerated by the helm for receiving incoming mail messages. If a clock skew is detected greater than this value, the message will be ignored. A check for skew can be disabled by setting `ok_skew=any`. The default value is 60 seconds.

5.4.8 The `other_override_var` Parameter

Optional. This parameter names a MOOS variable the helm will regard as being synonymous with the two default variables accepted for manual override, `MOOS_MANUAL_OVERRIDE`, and the legacy mispelling of this variable, `MOOS_MANUAL_OVERIDE`.

5.4.9 The `start_in_drive` Parameter

Optional. This parameter is set to either `true` or `false`. The default is `false` as the helm normally starts in PARK and needs to receive MOOS mail on the variable `MOOS_MANUAL_OVERRIDE` with the value of this variable set to `false`. When `start_in_drive` is set to `true`, the helm is in the DRIVE state upon start-up. The issue of helm state was discussed in more detail in Section 5.2.

5.4.10 The `hold_on_app` Parameter

Optional. Available after Release 17.7.x. This parameter names one or more MOOS apps that the helm will monitor in the `DB_CLIENTS` list. Once it has seen each named app at least once, the helm will release all mail generated by behaviors through their `onHelmStart()` function.

This may be useful for behaviors that produce configuration information to other support applications. Examples include the `pBasicContactMgr` and `pTaskManager` application. These apps need to be told by the behaviors the nature of alerts that may be generated, potentially resulting in spawned behaviors. If the behaviors produce multiple postings of the same MOOS variable, and those postings occur before the intended recipient apps have come on line (connected to the MOOSDB), then they may only receive the last piece of mail. With this utility, the helm will wait until all named apps are connected before posting the `onHelmStart()` mail.

The parameter may take a comma-separated list of apps, or multiple lines may be provided. The following two styles are equivalent:

```
hold_on_app = pBasicContactMgr, pTaskManager
and
hold_on_app = pBasicContactMgr
hold_on_app = pTaskManager
```

5.4.11 The `verbose` Parameter

Optional. This parameter affects how much information is written to the terminal on each iteration of the helm. The possible values are `verbose`, `terse`, or `quiet`. The `verbose` setting will write a brief helm report to the terminal on each iteration. With the `terse` setting minimal output will be
produced, a '*' character when not producing helm commands, and a '$' character when active and healthy. With the quiet setting, no output at all will be written to the terminal. The default value is terse. This setting can be changed after the helm is started by changing the value of HELM_VERBOSE in the MOOSDB.

5.4.12 An Example pHelmIvP MOOS Configuration Block

Below is an example configuration block for the helm.

Listing 5.2: An example pHelmIvP configuration block.

```plaintext
1 //-------- pHelmIvP configuration block -------------
2 ProcessConfig = pHelmIvP
3 {
4    AppTick = 4 // Defined for all MOOS processes
5    CommsTick = 4 // Defined for all MOOS processes
6
7    domain = course:0:359:360
8    domain = speed:0:3:16
9    domain = depth:0:500:101
10
11    behaviors = foobar.bhv
12    verbose = terse
13    ok_skew = ANY
14
15    start_in_drive = false
16    allow_park = true
17    park_on_allstop = false
18 }
```

The AppTick and CommsTick parameters are defined for all MOOS processes (see [47]) and specify the frequency in which the helm process iterates and communicates with the MOOSDB. The community parameter is not included in the configuration block because it is specified at the global level in the mission file.

5.5 Launching the Helm and Output to the Terminal Window

The helm can be launched either directly from the command line, or from within Antler. On the command line the usage is as follows:

```
Usage: pHelmIvP file.moos [file.bhv]...[file.bhv]
        [--help|-h] [--version|-v]
```

[file.moos] Filename to get MOOS config parameters.
[file.bhv] Filename to get IvP Helm config parameters.
[-v] Output version number and exit.
[-h] Output this usage information and exit.

If no behavior file is specified in the .moos file then a behavior file must be given on the command line. Multiple behavior files may be provided. Order of the arguments do not matter - command line arguments ending in .bhv will be read as behavior files, and those ending with .moos as MOOS files. The specification of behavior files may also be split between references in the .moos file and the command line. The duplicate specification of a single file will simply be ignored. Typical start-up output to the terminal is shown in Listing 3 below.
Listing 5.3: Example start-up output generated by the pHelmIvP process.

The output in lines 0-13 are standard output generated by a MOOS process launched and successfully connected to a running MOOSDB. Lines 15-30 are start-up output generated unique to the helm and the particular user usage. Behaviors used by the helm are either static or dynamic. Static behaviors are compiled in to the pHelmIvP executable. Dynamic behaviors are brought in at run time via shared libraries compiled separately. The helm looks for an environment variable $IVP_BEHAVIOR_DIRS$ for a colon-separated list of directories to search for shared libraries. If this variable is not set, or if one or more of the directories are not legitimate directories, an error message will indicate so between what is otherwise line 16 and 18 in Listing 3. This kind of error may not actually be problematic if the behaviors specified in the behavior file can all be otherwise successfully found.

For each specified behavior file, the information shown in lines 20-26 is generated to the terminal. For each behavior configuration in a given .bhv file, a single line is output as in lines 22-25 indicating that the behavior type is recognized and it is configured properly. A single unrecognized behavior or improper configuration will result in (a) an error message indicating the offending line number and file name, (b) the output of the actual offending line, and (c) immediate disconnection of the process from the MOOSDB and exit. (Tip: If the helm is launched with Antler an error during start-up will result in the closing of the pHelmIvP console window which makes it hard to catch useful error output for debugging. In this case, the helm should just be launched outside of Antler in its own terminal window.)

The output on line 31 of Listing 3, a series of dollar sign characters, indicates for each character,
the completion of a single helm iteration - a heartbeat output. This is the output when the `verbose` parameter is set to the default setting of `terse`. When set to `quiet` no output is generated at all. When set to `verbose`, a short multi-line report is generated for each iteration. An example is shown below in Listing 4:

```
Listing 5.4: An example helm iteration report generated by an active helm.
1 Iteration: 161 ******************************************
2 Helm Summary ---------------------------
3 loiter_a did NOT produce an obj-function
4 loiter_b produces obj-function - time:0.00 pcs: 9.00000 pwt: 100.00000
5 waypt_return did NOT produce an obj-function
6 loiter_timer did NOT produce an obj-function
7 Number of Objective Functions: 1
8 DESIRED_SPEED: 2.10
9 DESIRED_COURSE: 145.00
10 (End) Iteration: 161 ******************************************
```

On each iteration the Helm Summary indicates which behaviors produced objective functions (lines 2-5), and for those that did, it indicates the CPU time needed to generate the function, the number of pieces in the piecewise linear IvP function, and its priority weight. Following this, the decision rendered for current iteration is output with one line per decision variable (lines 7-8). This is a very thin summary of what is going on within the helm and it should be noted that the `uHelmScope` tool is a much better suited for monitoring helm activity and debugging.

5.6 Publications and Subscriptions for IvP Helm

The IvP Helm, like any MOOS process, can be specified in terms of its interface to the MOOSDB, i.e., what variables it publishes and what variables it subscribes for. It is impossible to provide a complete specification here since the helm is comprised of behaviors, and the means to include any number of third party behaviors. Each behavior is able to post variable-value pairs, published to the MOOSDB by the helm on behalf of the behavior at the end of the iteration. Likewise, each behavior may declare to the helm any number of MOOS variables it would like the helm to register for on its behalf. Barring these variables, published and subscribed for by the helm on behalf of individual behaviors, this section addresses the remaining portion of the helm’s publish - subscribe interface.

5.6.1 Variables published by the IvP Helm

Variables published by the IvP Helm are summarized below.

- **IVPHELM_SUMMARY**: Produced on each iteration of the helm for consumption by the `uHelmScope` application, [22]. It contains information on the current helm iteration regarding the number of IvP functions created, create time, solve time, which behaviors are active, running, idle, and the decision ultimately produced during the iteration. The summary does not include every component in each summary. Components that have not changed in value since the prior summary are dropped from the present summary. This is motivated by the goal to reducing the log file footprint for the helm.
- **IVPHELM_STATEVARS**: Produced periodically by the helm for consumption by the `uHelmScope` application, [22]. It contains a comma-separated list of MOOS variables involved in preconditions of any behavior, i.e., variables affecting behavior run states.
• **IVPHELM_DOMAIN**: Produced once by the helm at start-up for consumption by the uHelmScope application. It contains the specification of the IvP Domain in use by the helm.

• **IVPHELM_MODESET**: Produced once by the helm at start-up for consumption by the uHelmScope application. It contains the specification of the Hierarchical Mode Declarations, if any, in use by the helm.

• **IVPHELM_STATE**: Written by the helm on each iteration of the pHelmIvP MOOS application, regardless of whether the helm is in the DRIVE state or not. (see Section 5.2). It is either "DRIVE" or "PARK". This is the recommended MOOS variable for regarding as a "heartbeat" indicator of the helm.

• **HELM_IPF_COUNT**: Produced on each iteration of the helm. It contains the number of IvP functions involved in the solver on the current iteration.

• **CREATE_CPU**: The CPU time in seconds used in total by all behaviors on the current iteration for constructing IvP functions.

• **LOOP_CPU**: The CPU time in seconds used by the IvP solver in the current helm iteration.

• **BHV_IPF**: The helm will publish this variable for each active behavior in the current iteration. It contains a string representation of the IvP function produced by the behavior. It is used for visualization by the uFunctionVis application, and for logging and later playback and analysis.

• **PLOGGER_CMD**: This variable is published with the below value to ensure that the pLogger application logs the .bhv file along with the other data log files and the .moos file.

"COPY_FILE_REQUEST = filename.bhv"

• **DESIRED_**: Each of the decision variables in the IvPDomain provided in the helm configuration will have a separate posting prefixed by DESIRED, as in DESIRED_SPEED. One exception is that the variable course will be converted to heading for legacy reasons.

• **BHV_WARNING**: Although this variable may never be posted, it is the default MOOS variable used when a behavior posts a warning. A warning may be harmless but deserves consideration.

• **BHV_ERROR**: Although this variable may never be posted, it is the default MOOS variable used when a behavior posts what it considers a fatal error - one that the helm will interpret as a request to generate the equivalent of ALL-STOP.

In addition to the above variables, the helm will post any variable-value pair on behalf of a behavior that makes the request. These include endflags, runflags, idleflag, activeflags and inactiveflags.

### 5.6.2 Variables Subscribed for by the IvP Helm

Variables subscribed for by the IvP Helm are summarized below:

• **MOOS_MANUAL_OVERRIDE**: When set to true, usually by a third-party application such as iRemote, of from a command-and-control communication, the helm may relinquish control. If the helm was configured with active_start = true, it will not relinquish control (this may be changed).

• **HELM_VERBOSE**: Affects the console output produced by the helm. Legal values are verbose, terse, or quiet. See Section 5.5.

• **HELM_MAP_CLEAR**: When received, the helm clears an internal map that is used to surpress repeated duplicate postings. See Section 5.8.
In addition to the above variables, the helm will subscribe for any variable-value pair on behalf of a behavior that makes the request. This includes, but is not limited to, variables involved in the condition and updates parameters available generally for all behaviors.

5.7 Using a Standby Helm

A standby helm refers to the launching of a second helm running alongside another otherwise normally-configured primary helm. This is done to mitigate the risk associated with the possible failure of the primary helm. Both helms are instances of the pHelmIVP MOOS application configured with a different mission and/or behaviors. Presumably the standby helm is configured with a simpler, more conservative set of behaviors focused on the safe recovery of a vehicle. Although provisions are generally made during vehicle operations to detect a missing helm heartbeat, in situations such as the operation of a UUV under ice, it is not acceptable to simply have a vehicle halt and come to the surface. An attempt should be made to execute a simpler mission to return the vehicle to a safe location for recovery.

Use of a standby helm is as simple as adding a second configuration block in the .moos configuration file. The Kilo mission in Section 33 demonstrates a mission using the standby helm. Declaring a second helm as a standby helm requires a single additional line of the form

\[
\text{STANDBY} = N
\]

where \(N\) is the number of seconds a standby helm will tolerate an absent primary helm heartbeat before taking control from away the primary helm. See Listing 1 for an example. Once a standby helm take-over is triggered, the take-over is irreversible.

5.7.1 Two Types of Helm Failure, the Causes, and Detection

What does a helm crash mean, and how might it happen? A crash is result of code that causes the process to quit unexpectedly and without warning. A line as simple as `assert(0);` in the code would be sufficient to replicate a crash, but the cause of a crash could be much more subtle due referencing memory not properly allocated and so on. The other type of helm failure is a helm that hangs. This refers to the scenario where the helm enters a piece of code that takes too long or never finishes execution. This could be as trivial as reaching the line `while(1);` somewhere in the code. Either type of failure is serious. Despite the fact that we have never experienced these failures in any field exercise on any vehicle, the sudden disappearance of the helm process should be considered and handled as gracefully as possible.

How is a helm failure detected? The helm produces a heartbeat message on each iteration by posting to the variable `IVPHELM_STATE`. If the helm is configured to iterate four times per second, suspicion of failure could begin anytime after a quarter of a second passes without a posting to this variable. Under typical helm CPU load with common standard behaviors, the quarter second interval should be sufficient to finish the iteration. There are additional factors to consider however. There may be other processes on the machine dominating the CPU, thus challenging the helm to do its work in the expected time. There may be a periodic behavior calculation, such as recalculating a long path of waypoints, that may cause a spike in CPU cycles needed by the behavior. As a rule of thumb, the interval of time with the absence of a heartbeat, should arguably be two seconds or
more before declaring a helm failure. This interval is directly configurable, as the parameter $N$, in the `standby=$N` configuration line.

5.7.2 Handling a Helm Crash with the Standby Helm

A helm crash is the easier of the two failure cases to handle. The process is simply gone and no longer publishes to the MOOSDB. Of course the other MOOS processes, including the standby helm, have no way of knowing simply through their MOOS mailbox whether or not the primary helm is gone or just delayed. In either case the sequence of events is the following:

1. The standby helm detects the absence of heartbeat for more than $N$ seconds.

2. On the very same iteration, the standby helm posts `IVPHELM_STATE = DRIVE+` rather than `IVPHELM_STATE = STANDBY` which it had been posting on every iteration up until now. It also begins posting a desired helm decision on this iteration.

The standby helm has all the same functionality as the primary helm, modulo the behavior and mission configuration. It will be in the `DRIVE` state only if not manually overridden. If `MOOS_MANUAL_OVERRIDE=true` when the standby helm takes over, it will be in initially in `PARK`, not `DRIVE`. It will post `IVPHELM_STATE=PARK+`. Note the standby helm will append the `+'` character to the helm state string to help other applications and the user discern that the helm state being posted is from a standby helm that has taken control.

The Kilo example mission in Section 33 walks through a simulated helm crash. The output of the helm(s) prior to and after the standby takeover is discussed in Figure 84.

5.7.3 Handling a Hung Helm with the Standby Helm

Handling a helm that has hung requires a bit more consideration. The tricky part is that quite possibly the hung helm is only temporarily hung, and at some point it will become unhung and may operate for a single iteration as if it is still the helm in charge. Two helms, with different missions, both thinking they are in charge! The sequence of events is summarized below:

1. The standby helm detects the absence of heartbeat for more than $N$ seconds.

2. On the very same iteration, the standby helm posts `IVPHELM_STATE = DRIVE+` rather than `IVPHELM_STATE = STANDBY` which it had been posting on every iteration up until now. It also begins posting a desired helm decision on this iteration.

3. Some time later, the original primary helm finishes its iteration and posts a helm decision, e.g., a set of postings to the helm decision variables, `DESIRED_HEADING` etc.

4. On the next iteration of the original primary helm it notices that another helm has posted a non-standby heartbeat, e.g., a posting to `IVPHELM_STATE` not equal to "STANDBY".

5. The original primary helm posts an all-stop and `IVPHELM_STATE = DISABLED`. It is the last time it will post a heartbeat or helm decision.

6. The standby helm continues to be in control posting helm decisions oblivious to the former primary helm’s epiphany and temporary influence.
The key issue in this scenario is that the original primary helm does indeed post a helm decision when it becomes un-hung even though the standby helm may have long ago taken over and may be posting a sequence of helm decisions completely at odds with the decision posted by the newly awakened un-hung primary helm.

No assumptions can be made about the MOOS process listening to the sequence of helm decisions. It may be payload interface process, or a native PID controller, or some other process responsible for converting helm decisions to lower level actuator commands. The assumption that is made here is that a one-time aberation in the sequence in helm decisions is tolerable. This aberation is not going to result in an actuator breaking due to a sudden change in the command sequence such as high-speed, zero-speed, high-speed.

It’s also worth noting that original primary helm, upon awakening and learning it is no longer in control, does indeed post one more helm decision, an all-stop decision. This is done to ensure that an all-stop decision, that may have been put into effect by the standby helm, is not superceded by the output of the original primary helm’s final helm decision made upon wakeup.

5.7.4 Activity of the Standby Helm While Standing By

In the standby state the helm will do nothing in its iterate loop other than post a heartbeat character to the terminal if configured to do so. It will not call on any behaviors to do anything. The helm will however read and process all of its otherwise subscribed for mail. In particular it will monitor for postings to IVPHELM\_STATE, and will initiate a take-over when it hasn’t received such mail for N seconds. Note the standby helm also publishes to this variable, IVPHELM\_STATE = STANDBY, but ignores mail originating from itself.

The helm will also read and process all other mail in its inbox, updating the helm’s information buffer. The helm’s information buffer is described in detail in Section 7.5.2. Note that the information buffer also keeps a history of postings to a particular variable so that normally a behavior may process multiple postings if multiple postings were made to the MOOSDB between helm iterations. This history is cleared by the helm at the end of each helm iteration. When the helm is in the standby state it will also clear the history after each iteration even though the behaviors have not been given the chance to access this history. This is to ensure that the information buffer history doesn’t grow without bound while in standby mode. For example, if the standby and primary helm are both configured with a waypoint behavior and the primary helm visits half the points at the point of a helm crash, the standby helm’s waypoint behavior would start with the first waypoint.

5.7.5 Activity of the Primary Helm After Take-Over

A primary helm that has been taken over is refered to as a disabled helm. It will post only once IVPHELM\_STATE = DISABLED. The helm process lives on however doing next to nothing. It does not read its mail, and the iterate loop simply posts a heartbeat character (’!’) to the terminal if configured to do so, with the helm configuration parameter verbose=terse.

5.8 Automated Filtering of Successive Duplicate Helm Publications

The helm implements a ”duplication filter” to drastically reduce the amount of mail posted by the helm on behalf of behaviors. This filter has been noted to reduce the overall log file size seen during
in-water exercises by 60-80%. Reductions at this level noticeably facilitate the use of post-mission analysis tools and data archiving. For the most part this filter is operating behind the scenes for the typical helm user. However, knowledge of it is indeed relevant for users wishing to implement their own behaviors, and we discuss it here to explain a bit what is behind the variable HELM_MAP_CLEAR to which the helm subscribes, and listed above in Section 5.6.2.

5.8.1 Motivation for the Duplication Filter

The primary motivation of implementing the duplication filter is to reduce the amount of unnecessary mail posted by the helm on behalf behaviors, and thereby greatly reduce the size of log files and facilitate the post-mission handling of data. By unnecessary we mean successive variable-value pairs that match exactly in both fields. Surely there are cases when a behavior developer may not want this filter, and there are simple ways to bypass the filter for any post. But in most cases, successive duplicate posts are just redundant and unnecessary.

5.8.2 Implementation and Usage of the Duplication Filter

The helm keeps two maps (STL maps in C++), one for string data and one for numerical data:

- **KEY --> StringValue**
- **KEY --> DoubleValue**

The two maps correspond to the double and string message types in MOOS (see Section 3.2). The **KEY** is typically the MOOS variable name. Inside a behavior implementation, the following four functions are available:

- `void postMessage(string varname, string value, string key="");`
- `void postMessage(string varname, double value, string key="");`
- `void postBoolMessage(string varname, bool value, string key="");`
- `void postIntMessage(string varname, double value, string key="");`

These functions are available in all behavior implementations because they are defined in the IvPBehavior superclass, of which all behaviors are subclasses. Before the helm posts a message to the MOOSDB the filter is applied by a simple check to its map to determine if there is a value match on the given key. If a match is made, the post will not be made to the MOOSDB on the behavior’s behalf. The `postIntMessage()` function is merely a convenience version of the `postMessage()` function that rounds the variable value to the nearest integer to further reduce posts when combined with the filter. The `postBoolMessage()` ultimately posts a string value "true" or "false".

The default value of the **key** parameter is the empty string, and in most cases this parameter can be omitted without disabling the duplication filter. This is because the **KEY** used by the caller is only part of the key actually used by the duplication filter. The actual key is the concatenation of (a) the behavior name, (b) the variable name, and (c) the key passed by the caller. Thus the default value, the empty string, still results in a decent key being used by the filter. The key is augmented by the behavior name because often there is more than one behavior posting messages on same variable. The optional key parameter is used for two reasons. First, it can be used to further distinguish posts within a behavior on the same variable name. Second, when the key value has the special value "repeatable", then no key is used and the duplication filter is disabled for that variable posting.
Two additional convenience functions are available:

```c
void postRepeatableMessage(string varname, string value);
void postRepeatableMessage(string varname, double value);
```

A posting of `postRepeatableMessage("FOO", 100)` is equivalent to `postMessage("FOO", 100, "repeatable").`

### 5.8.3 Clearing the Duplication Filter

Occasionally a user, or another MOOS application in the same community as the helm, may want to "clear" the map used by the helm to implement its duplication filter. This can be done by writing to variable `HELM_MAP_CLEAR`, with any value. This may be necessary for the following reason. Suppose a GUI application subscribes for the variable `VIEW_SEGLIST` which contains a list of line segments for rendering. If the viewer application is launched after the variable is published, the application will only receive the most recent mail on the variable `VIEW_SEGLIST`. There may be publications to this variable, made prior to the most recent publication, that are relevant to the GUI application at launch time. Those publications for the variable `VIEW_SEGLIST` may not be the most recent from the perspective of the `MOOSDB`, but they may be the most recent from the perspective of a particular behavior in the helm. By clearing the filter, it gives each behavior the chance to once again have all of its variable-value posts made to the `MOOSDB`. In the `pMarineViewer` application, a publication to `HELM_MAP_CLEAR` is made upon start-up. Clearing the filter will only clear the way for the next post for a given variable. It will not result in the publishing to the `MOOSDB` of the contents of the maps used by the filter.
6 IvP Helm Autonomy

6.1 Overview

An autonomous helm is primarily an engine for decision making. The IvP Helm uses a behavior-based architecture to organize its decision making and is distinctive in the manner in which it resolves competition between competing behaviors - it performs multi-objective optimization on their collective output using a mathematical programming model called interval programming. Here the IvP Helm architecture is described and the means for configuring it given a set of behaviors and a set of mission objectives.

6.1.1 The Influence of Brooks, Stallman and Dantzig on the IvP Helm

The notion of a behavior-based architecture for implementing autonomy on a robot or unmanned vehicle is most often attributed to Rodney Brooks’ Subsumption Architecture, [38]. A key principle at the heart of Brooks’ architecture and arguably the primary reason its appeal has endured, is the notion that autonomy systems can be built incrementally. Notably, Brooks’ original publication pre-dated the arrival of Open Source software and the Free Software Foundation founded by Richard Stallman. Open Source software is not a pre-requisite for building autonomy systems incrementally, but it has the capability of greatly accelerating that objective. The development of complex autonomy systems stands to significantly benefit if the set of developers at the table is large and diverse. Even more so if they can be from different organizations with perhaps even the loosest of overlap in interest regarding how to use the collective end product.

As discussed in Section 2.5, a key issue in behavior-based autonomy has been the issue of action selection, and the IvP Helm is distinct in this regard with the use of multi-objective optimization and interval programming. The algorithm behind interval programming, as well as the term itself, was motivated by the mathematical programming model, linear programming, developed by George Dantzig, [41]. The key idea in linear programming is the choice of the particular mathematical construct that comprises an instance of a linear programming problem - it has enough expressive flexibility to represent a huge class of practical problems, and the constructs can be effectively exploited by the simplex method to converge quickly even on very large problem instances. The constructs used in interval programming to represent behavior output (piecewise linear functions) were likewise chosen to have enough expressive flexibility to handle any current and future behavior, and due to the opportunity to develop solution algorithms that exploit the piecewise linear constructs.

6.1.2 Traditional and Non-traditional Aspects of the IvP Behavior-Based Helm

The IvP Helm indeed takes its motivation from early notions of the behavior-based architecture, but is also quite different in many regards. The notion of behavior independence to temper the growth of complexity in progressively larger systems is still a principle closely followed in the IvP Helm. Behaviors may certainly influence one another from one iteration to the next, as we’ll see in discussions in this section. This was also evident in the Alpha example mission in Section 4 where the completion of the Survey behavior triggered the Return behavior. But within a single iteration, the output generated by a single behavior is not affected at all by what is generated by other behaviors in the same iteration. The only inter-behavior "communication" realized within an iteration comes when the IvP solver reconciles the output of multiple behaviors. The independence of behaviors not
only helps a single developer manage the growth of complexity, but it also limits the dependency between developers. A behavior author need not worry that a change in the implementation of another behavior by another author requires subsequent recoding of one’s own behavior(s).

Certain aspects of behaviors in the IvP Helm may also be a departure from some notions traditionally associated (fairly or not) with behavior-based architectures:

- Behaviors have state. IvP behaviors are instances of a class with a fairly simple interface to the helm. Inside they may be arbitrarily complex, keep histories of observed sensor data, and may contain algorithms that could be considered "reactive" or "plan-based".

- Behaviors influence each other between iterations. The primary output of behaviors is their objective function, ranking the utility of candidate actions. IvP behaviors may also generate variable-value posts to the MOOSDB observable by behaviors on the next helm iteration. In this way they can explicitly influence other behaviors by triggering or suppressing their activation or even affecting the parameter configuration of other behaviors.

- Behaviors may accept externally generated plans. The input to a behavior can be anything represented by a MOOS variable, and perhaps generated by other MOOS processes outside the helm. It is allowable to have one or more planning engines running on the vehicle generating output consumed by one or more behaviors.

- Several instances of the same behavior. Behaviors generally accept a set of configuration parameters that allow them to be configured for quite different tasks or roles in the same helm and mission. Different waypoint behaviors, for example, can be configured for different components of a transit mission. Or different collision avoidance behaviors can be instantiated for different contacts.

- Behaviors can be run in a configurable sequence. Due to the condition and endflag parameters defined for all behaviors, a sequence of behaviors can be readily configured into a larger mission plan.

- Behaviors rate actions over a coupled decision space. IvP functions generated by behaviors are defined over the Cartesian product of the set of vehicle decision variables. This is distinct from the de-coupled decision making style proposed in [49] and [52] - early advocates of multi-objective optimization in behavior-based action selection.

6.1.3 Two Layers of Building Autonomy in the IvP Helm

The autonomy in play on a vehicle during a particular mission is the product of two distinct efforts - (1) the development of vehicle behaviors and their algorithms, and (2) mission planning via the configuration of behaviors and mode declarations. The former involves the writing of new source code, and the latter involves the editing of mission behavior files, such as the simple example for the Alpha example mission in Listing 1.
6.2 Inside the Helm - A Look at the Helm Iterate Loop

Like other MOOS applications, the IvP Helm implements an Iterate() loop within which the basic function of the helm is executed. Components of the Iterate() loop, with respect to the behavior-based architecture, are described in this section. The basic flow, in five steps, is depicted in Figure 13. Description of the five components follow.

Figure 13: The pHelmIvP Iterate Loop: (1) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors, (2) If there were any mode declarations in the mission behavior file they are evaluated at this step. (3) Each behavior is queried for its contribution and may produce an IvP function and a list of variable-value pairs to be posted to the MOOSDB at the end of the iteration, (4) the objective functions are resolved to produce an action, expressible as a set of variable-value pairs, (5) all variable-value pairs are published to the MOOSDB for other MOOS processes to consume.

6.2.1 Step 1 - Reading Mail and Populating the Info Buffer

The first step of a helm iteration occurs outside the Iterate() loop. As depicted in Figure 5, a MOOS application will read its mail by executing its OnNewMail() function just prior to executing its Iterate() loop if there is any mail in its in-box. The helm parses mail to maintain its own information buffer which is also a mapping of variables to values. This is done primarily for simplicity - to ensure that each behavior is acting on the same world state as represented by the info buffer. Each behavior has a pointer to the buffer and is able to query the current value of any variable in the buffer, or get a list of variable-value changes since the previous iteration.
6.2.2 Step 2 - Evaluation of Mode Declarations

Once the information buffer is updated with all incoming mail, the helm evaluates any mode declarations specified in the behavior file. Mode declarations are discussed in Section 6.4. In short, a mode is represented by a string variable that is reset on each iteration based on the evaluation of a set of logic expressions involving other variables in the buffer. The variable representing the mode declaration is then available to the behavior on the current iteration when it, for example, evaluates its condition parameters. A condition for behavior participating in the current iteration could therefore read something like condition = (MODE==SURVEYING). The exact value of the variable MODE is set during this step of the Iterate() loop.

6.2.3 Step 3 - Behavior Participation

In the third step much of the work of the helm is realized by giving each behavior a chance to participate. Each behavior is queried sequentially - the helm contains no separate threads in this regard. The order in which behaviors is queried does not affect the output. This step contains two distinct parts for each behavior - (1) Determination of whether the behavior will participate, and (2) production of output if it is indeed participating on this iteration. Each behavior may produce two types of information as the Figure 13 indicates. The first is an objective function (or "utility" function) in the form of an IvP function. The second kind of behavior output is a list of variable-value pairs to be posted by the helm to the MOOSDB at the end of the Iterate() loop. A behavior may produce both kinds of information, neither, or one or the other, on any given iteration.

6.2.4 Step 4 - Behavior Reconciliation

In the fourth step depicted in Figure 13, the IvP functions are collected by the IvP solver to produce a single decision over the helm’s decision space. Each function is an IvP function - an objective function that maps each element of the helm’s decision space to a utility value. In this case the functions are of a particular form - piecewise linearly defined. That is, each piece is an interval of the decision space with an associated linear function. Each function also has an associated weight and the solver performs multi-objective optimization over the weighted sum of functions (in effect a single objective optimization at that point). The output is a single optimal point in the decision space. For each decision variable the helm produces another variable-value pair, such as DESIRED_SPEED = 2.4 for publication to the MOOSDB.

6.2.5 Step 5 - Publishing the Results to the MOOSDB

In the last step, the helm simply publishes all variable-value pairs to the MOOSDB, some of which were produced directly by the behaviors, and some of which were generated as output from the IvP Solver. The helm employs the duplication filter described in Section 5.8, only on the variable-value pairs generated directly from the behaviors, and not the variable-value pairs generated by the IvP solver that represent a decision in the helm’s domain. For example, even if the decision about a vehicle’s depth, represented by the variable DESIRED_DEPTH produced by the helm were unchanged for 5 minutes of operation, it would be published on each iteration of the helm. To do otherwise could give the impression to consumers of the variable that the variable is "stale", which could trigger an unwanted override of the helm out of concern for safety.
6.3 Mission Behavior Files

The helm is configured for a particular mission primarily through one or more mission behavior files, typically with a *.bhv suffix. Behavior files have three types of entries, usually but not necessarily kept in three distinct parts - (1) variable initializations, (2) behavior configurations, and (3) hierarchical mode declarations. These three parts are discussed below. The example alpha.bhv file in Listing 1 did not contain hierarchical mode declarations, but does contain examples of variable initializations and behavior configurations.

6.3.1 Variable Initialization Syntax

The syntax for variable initialization is fairly straight-forward:

```
initialize <variable> = <value>
...
initialize <variable> = <value>
```

Multiple initializations may be declared on a single line by separating each variable-value pair with a comma. The keyword `initialize` is case insensitive. The `<variable>` is indeed case sensitive since it will be published to the MOOSDB and MOOS variables are case sensitive when registered for by a client. The `<value>` may or may not be case sensitive depending on whether or not a client registering for the variable regards the case. Considering again the helm Iterate() loop depicted in Figure 13, variable initializations are applied to the helm’s information buffer prior to the very first helm iteration, but are posted to the MOOSDB at the end of the first helm iteration.

By default, an initialization will overwrite any prior value posted to the MOOSDB. There may be situations, however, where the user’s desired effect is that the initialization only be applied if no other value has yet been written to the given MOOS variable. The syntax in this case would be:

```
initialize_ <variable> = <value> // Deferring to prior posts if any
```

By using the ”underscore” version of the `initialize` declaration, the helm will first register with the MOOSDB for the given variable, wait an iteration until it has had chance to receive mail from the MOOSDB on that variable, and only initialize the variable if nothing is otherwise known about that variable. (Note to the very discerning reader: Such an initialization also includes both an update to the helm’s information buffer and a post to the MOOSDB. Posts to the MOOSDB by the helm, as part of a variable initialization, will indeed show up in the helm’s incoming mailbox on the next iteration, but they are tagged in such a way as to be ignored by the helm. This is to ensure that they do not ”collide” with posts made by other processes.)

6.3.2 Behavior Configuration Syntax

The bulk of the helm configuration is done with individual behavior parameter blocks which have the following form:

```
Behavior = <behavior-type>
```
The first line is a declaration of the behavior type. The keyword \texttt{Behavior} is not case sensitive, but
the \texttt{<behavior-type>} is. This is followed by an open brace on a separate line. Each subsequent line
sets a particular parameter of the behavior to a given value. The behavior configuration concludes
with a close brace on a separate line. The issue of case sensitivity for the \texttt{<parameter>} and \texttt{<value>}
entries is a matter determined by the individual behavior implementation.

As a convention (not enforced in any way) general behavior parameters, defined at the IvP
Behavior superclass level, are grouped together and listed before parameters that apply to a specific
behavior. For example, in the Alpha example in Listing 1, the general behavior parameters are
listed on lines 8-12 and 22-25, but the parameters specific to the waypoint behavior, \texttt{speed}, \texttt{radius},
and \texttt{points}, follow in a separate block. Generally it is not mandatory to provide a parameter-value
pair for each parameter defined for a behavior, given that meaningful defaults are in place within
the behavior implementation. Some parameters are indeed mandatory however. Documentation for
the individual behavior should be consulted. Multiple instances of a behavior type are allowed, as
in the Alpha example where there are two waypoint behaviors - one for traversing a set of points,
and one for returning to a vehicle recovery point. Each behavior should have its own unique value
provided in the \texttt{name} parameter.

6.3.3 Hierarchical Mode Declaration Syntax

Hierarchical Mode Declarations are covered in depth in Section 6.4, but the syntax is briefly discussed
here. A behavior file contains a set of declaration blocks of the form:

\begin{verbatim}
Set <mode-variable-name> = <mode-value>
{
    <mode-variable-name> = <parent-value>
    <condition>
    ...
    <condition>
} <else-value>
\end{verbatim}

A tree will be formed where each node in the tree is described from the above type of declaration.
The keyword \texttt{Set} is case insensitive. The \texttt{<mode-variable-name>}, \texttt{<parent-value>} and \texttt{<else-value>}
are case sensitive. The \texttt{<condition>} entries are treated exactly as with the \texttt{condition} parameter for
behaviors, see Section 6.5.1.

As indicated in Figure 13, the value of each mode variable is reset at the outset of the \texttt{Iterate()} loop,
after the information buffer is updated with incoming mail. A mode variable is set by
progressing through each declaration block, and determining whether the conditions are met. Thus
the ordering of the declaration blocks is significant - the specification of parent should be made
prior to that of a child. Examples are further discussion can be found below in Section 6.4.
6.4 Hierarchical Mode Declarations

Hierarchical mode declarations (HMDs) are an optional feature of the IvP Helm for organizing the behavior activations according to declared mission modes. Modes and sub-modes can be declared, in line with a mission planner’s own concept of mission evolution, and behaviors can be associated with the declared modes. In more complex missions, it can facilitate mission planning (in terms of less time and better detection of human errors), and it can facilitate the understanding of exactly what is happening in the helm - during the mission execution and in post-analysis.

6.4.1 Background

A trend of unmanned vehicle usage can be characterized as being increasingly less of the shorter, scripted variety to be increasingly more of the longer, adaptive mission variety. A typical mission in our own lab five years ago would contain a certain set of tasks, typically waypoints and ultimately a rendezvous point for recovering the vehicle. Data acquired during deployment was off-loaded and analyzed later in the laboratory. What has changed? The simultaneous maturation of acoustic communications, on-board sensor processing, and longer vehicle battery life has dramatically changed the nature of mission configurations. The vehicle is expected to adapt to both the phenomena it senses and processes on board, as well as adapt its operation given field-control commands received via acoustic, radio or satellite communications. Multi-vehicle collaborative missions are also increasingly viable due to lower vehicle costs and mature acomms capabilities. In such cases a vehicle is not only adapting to sensed phenomena and field commands, but also to information from collaborating vehicles.

Our missions have evolved from having a finite set of fixed tasks to be composed instead of a set of modes, an initial mode when launched, an understanding of what brings us from one mode to another, and what behaviors are in play in each mode. Modes may be entered and exited any number of times, in exact sequences unknown at launch time, depending on what they sense and how they are commanded in the field.

6.4.2 Behavior Configuration Without Hierarchical Mode Declarations

Behaviors can be configured for a mission without the use of hierarchical mode declarations - support for HMDs is a relatively recent addition to the helm. HMDs are a tool for organizing which behaviors are idle or participating in which circumstances. Consider the alpha example mission in Section 4, and the behavior file in Listing 1. By examination of the behavior file, and experimenting a bit with the viewer during simulation, the vehicle apparently is always in one of three modes - (a) idle, (b) surveying the waypoints, or (c) returning to the launch point. This is achieved by the condition parameters for the two behaviors. There are only two variables involved in the behavior conditions, DEPLOY and RETURN. If restricted to Boolean values, the below table confirms the observation that there are only three possible modes.
There are a couple drawbacks with this however. First, the modes are to be inferred from the behavior conditions and this is not trivial in missions with larger behavior files. Mapping the behavior conditions to a mode is useful both in mission planning and mission monitoring. In the alpha mission, in order to understand at any given moment what mode the vehicle is in, the two variables need to be monitored, and the above table internalized. The second drawback is the increased likelihood of error, in the form of unintentionally being in two modes at the same time, or being in an undefined mode. For example, line 11 in Listing 1 really should read \texttt{RETURN \neq true}, and not \texttt{RETURN = false}. Since there is no Boolean type for MOOS variables, this variable could be set to "False" and the condition as it reads on line 11 in Listing 1 would not be satisfied, and the vehicle would be in the idle state, despite the fact that \texttt{DEPLOY} may be set to \texttt{true}. These problems are alleviated by the use of hierarchical mode declarations.

### Syntax of Hierarchical Mode Declarations - The Bravo Mission

An example is provided showing of the use of hierarchical mode declarations by extending the Alpha mission described in Section 4. This example mission is dubbed the "Bravo" mission in the directory \texttt{s2_bravo} alongside the Alpha mission \texttt{s1_alpha} in the MOOS-IvP distribution (Section 4.1). It is also given fully in Listing 1 on the next page. The \textit{implicit} modes of the Alpha mission, described in Table 3, are explicitly declared in the Bravo behavior file to form the following hierarchy:

![Hierarchical modes for the Bravo mission](image)
The hierarchy in Figure 14 is formed by the mode declaration constructs on the left-hand side, taken as an excerpt from the `bravo.bhv` file. After the mode declarations are read when the helm is initially launched, the hierarchy remains static thereafter. The hierarchy is associated with a particular MOOS variable, in this case the variable `MODE`. Although the hierarchy remains static, the mode is re-evaluated at the outset of each helm iteration based on the conditions associated with nodes in the hierarchy. The mode evaluation is represented as a string in the variable `MODE`. As shown in Figure 14 the variable is the concatenation of the names of all the nodes. The mode evaluation begins sequentially through each of the blocks. At the outset the value of the variable `MODE` is reset to the empty string. After the first block in Figure 14 `MODE` will be set to either "Active" or "Inactive". When the second block is evaluated, the condition "MODE=Active" is evaluate based on how `MODE` was set in the first block. For this reason, mode declarations of children need to be listed after the declarations of parents in the behavior file.

Once the mode is evaluated, at the outset of the helm iteration, it is available for use in the conditions of the behaviors, as in lines 20 and 23 in Listing 1. Note the "==" relation in lines 18 and 36. This is a string-matching relation that matches when one side matches exactly one of the components in the other side’s colon-separated list of strings. Thus "Active" == "Active:Returning", and "Returning" == "Active:Returning". This is to allow a behavior to be easily associated with an internal node regardless of its children. For example if a collision-avoidance behavior were to be added to this mission, it could be associated with the "Active" mode rather than explicitly naming all the sub-modes of the "Active" mode.


```plaintext
1 initialize DEPLOY = false
2 initialize RETURN = false
3
4 //----------------- Declaration of Hierarchical Modes
5 set MODE = ACTIVE {
6    DEPLOY = true
7 } INACTIVE
8
9 set MODE = SURVEYING {
10    MODE = ACTIVE
11    RETURN != true
12 } RETURNING
13
14 //-------------------------------
15 Behavior = BHV_Waypoint
16 {
17    name = waypt_survey
18    pwt = 100
19    condition = MODE == SURVEYING
20    endflag = RETURN = true
21    perpetual = true
22
23    lead = 8
24    lead_damper = 1
25    speed = 2.0 // meters per second
26    radius = 4.0
27    nm_radius = 10.0
28    points = 60,-40;60,-160;150,-160;180,-100;150,-40
29    repeat = 1
30 }
31
32 //-------------------------------
```
6.4.4 A More Complex Example of Hierarchical Mode Declarations

The Bravo example given above, while having the benefit of being a working example distributed with the codebase, is not complex. In this section a modestly complex, although fictional, hierarchy is provided to highlight some issues with the syntax. The hierarchy with the corresponding mode declarations are shown in Figure 15. The declarations are given in the order of layers of the tree ensuring that parents are declared prior to children. As with the Bravo example in Figure 14, the nodes that represent realizable modes are depicted in the darker (green) color.

![Diagram of hierarchical mode declarations](image)

Figure 15: Example Hierarchical Mode Declaration: The hierarchy on the right is constructed from the set of mode declarations on the left (with fictional conditions). Darker nodes represent modes that are realizable through some combination of conditions.

The "Alpha" mode for example is not realizable since it has the children "Delta" and "Echo", with the latter being set as the <else-value> if the conditions of the former at not met. The "Bravo"
mode is realizable since it has no children. The "Echo" mode is realizable despite having children because the "Tango" mode is not the <else-value> of the "Sierra" mode declaration. For example, if the following three conditions hold, (a) "MISSION=SURVEYING", (b) "SITE!=Archipelagos", and (c) "WATER DEPTH=Medium", then the value of the variable MODE would be set to "Alpha:Echo". Finally, note that the condition in the "Sierra" declaration, "MODE=Alpha:Echo", is specified fully, i.e., "MODE=Echo" would not achieve the desired result.

6.4.5 Monitoring the Mission Mode at Run Time

The mission mode can be monitored at run time in a couple ways. First, since the mode variable is posted as a MOOS variable, any MOOS scope tool will work, e.g., uXMS, uMS, uHelmScope. Using uHelmScope, the mission variable can be monitored as part of the basic MOOSDB scoping capability, but it is also displayed as part of the uHelmScope app, and in the AppCasting output of pHelmIVP.

The uHelmScope tool also has a mode in which the entire mode hierarchy may be rendered - solely to provide a visual confirmation that the hierarchy specified with the mode declarations in the behavior file does in fact correspond to what the user intended. Currently there are no tools to automatically render the mode hierarchy in a manner like the right hand side of Figure 15. The uHelmScope output for the example in Figure 15 is shown in listing 2 below.

Listing 6.2: The mode hierarchy output from uHelmScope for the example in Figure 15.

```
1 ModeSet Hierarchy:
2 ----------------------------------------------
3 Alpha
4   Delta
5   Echo
6   Sierra
7   Tango
8 Bravo
9 Charlie
10 Foxtrot
11 Golf
12 ----------------------------------------------
13 CURRENT MODE(S): Charlie:Foxtrot
14
15 Hit 'r' to resume outputs, or SPACEBAR for a single update
```

More on this feature of the uHelmScope can be found in the uHelmScope documentation, [22]. It’s worth noting that poking the value of a mode variable will have no effect on the helm operation. The mission mode cannot be commanded directly. The mode variable is reset at the outset of the helm iteration, and the helm doesn’t even register for mail on mode variables.

6.5 Behavior Participation in the IvP Helm

The primary work of the helm comes when the behaviors participate and do their thing, at each round of the helm Iterate() loop. As depicted in Figure 13, once the mode has been re-evaluated taking into consideration newly received mail, it is time for the behaviors (well, some at least) to step up and do their thing.
6.5.1 Behavior Run Conditions

On any single iteration a behavior may participate by generating an objective function to influence the helm’s output over its decision space. Not all behaviors participate in this regard, and the primary criteria for participation is whether or not it has met each of its ”run conditions”. These are the conditions laid out in the behavior file of the form:

```
condition = <logic-expression>
```

The `<logic-expression>` syntax is described in Appendix A. Conditions are built from simple relational expressions, the comparison of MOOS variables to specified literal values, or the comparison of MOOS variables to one another. Conditions may also involve Boolean logic combinations of relation expressions. A behavior may base its conditions on any MOOS variable such as:

```
condition = (DEPLOY=true) and (STATION_KEEP != true)
```

A run condition may also be expressed in terms of a helm mode, as described in the next Section 6.5.2 such as:

```
condition = (MODE == LOITERING)
```

All MOOS variables involved in run condition expressions are automatically subscribed for by the helm to the MOOSDB.

6.5.2 Behavior Run Conditions and Mode Declarations

The use of hierarchical mode declarations potentially simplify the expressions used as run conditions. The conditions in practice could be limited to:

```
condition = <mode-variable> = <mode-value>, or
condition = <mode-variable> == <mode-value>.
```

Conditions were used in this way with the Bravo mission in Listing 1, as an alternative to their usage in the Alpha mission example in Listing 1.

Note the use of the double-equals relation above. This relation is used for matching against the strings used to represent the hierarchical mode. The two strings match if the ordered components of one side are a subset of the ordered components of the other. Components are colon-separated. For example, using the illustrative hierarchy from Figure 15:

```
"Alpha:Echo:Sierra" == "Sierra"
"Alpha:Echo:Sierra" == "Echo:Sierra"
"Alpha:Echo:Sierra" == "Alpha"
"Sierra" == "Alpha:Echo:Sierra"
"Charlie:Foxtrot" == "Charlie:Foxtrot"

"Alpha:Echo:Sierra" != "Alpha:Sierra"
```
6.5.3 Behavior Run States

On any given helm iteration a behavior may be in one of four states depicted in Figure 16:

- **Idle**: A behavior is idle if it is not complete and it has not met its run conditions as described above in Section 6.5.1. The helm will invoke an idle behavior’s `onIdleState()` function.

- **Running**: A behavior is running if it has met its run conditions and it is not complete. The helm will invoke a running behavior’s `onRunState()` function thereby giving the behavior an opportunity to contribute an objective function.

- **Active**: A behavior is active if it is running and it did indeed produce an objective function when prompted. There are a number of reasons why a running behavior may not be active. For example, a collision avoidance behavior where the object of the behavior is sufficiently far away.

- **Complete**: A behavior is complete when the behavior itself determines it to be complete. It is up to the behavior author to implement this, and some behaviors may never complete. The function `setComplete()` is defined generally at the behavior superclass level, for calling by a behavior author. This provides some some standard steps to be taken upon completion, such as posting of endflags, described below in Section 6.5.4. Once a behavior is in the complete state, it remains in that state permanently. All behaviors have a `duration` parameter defined to allow it to be configured to time-out if desired. When a time-out occurs the behavior state will be set to complete.

6.5.4 Behavior Flags and Behavior Messages

Behaviors may post some number of messages, i.e., variable-value pairs, on any given iteration (see Figure 13). These message can be critical for coordinating behaviors with each other and to other MOOS processes. The can also be invaluable for monitoring and debugging behaviors configured for particular missions. To be more accurate, behaviors don’t post messages to the MOOSDB, they request the helm to post messages on its behalf. The helm collects these requests and publishes them to the MOOSDB at the end of the `Iterate()` loop. It also filters them for successive duplicates as discussed in Section 5.8.

There is a standard method, configurable in the behavior file, for posting messages based on the run state of the behavior. These are referred to as behavior flags, and there are several types, `endflag`, `idleflag`, `runflag`, `activeflag`, `inactiveflag`, and `spawnflag`. The variable-value pairs representing each flag are set in the behavior file for the corresponding behavior. See line 11 in Listing 1 for example.
• **endflag**: An endflag is posted once when or if the behavior enters the complete state. The variable-value pair representing the endflag is given in the endflag parameter in the behavior file. Multiple endflags may be configured for a behavior.

• **idleflag**: An idleflag is posted by the helm when the behavior enters the idle state. The variable-value pair representing the idleflag is given in the idleflag parameter in the behavior file. Multiple idleflags may be configured for a behavior.

• **runflag**: A runflag is posted by the helm when the behavior enters the running state from the idle state. A runflag is posted exactly when an idleflag is not. The variable-value pair representing the runflag is given in the runflag parameter in the behavior file. Multiple runflags may be configured for a behavior.

• **activeflag**: An activeflag is posted by the helm when the behavior enters the active state. The variable-value pair representing the activeflag is given in the activeflag parameter in the behavior file. Multiple activeflags may be configured for a behavior.

• **inactiveflag**: An inactiveflag is posted by the helm when the behavior enters a state that is not the active state. The variable-value pair representing the inactiveflag is given in the inactiveflag parameter in the behavior file. Multiple inactiveflags may be configured for a behavior.

• **spawnflag**: An spawnflag is posted by the helm when templated behavior is spawned. Examples include the collision and obstacle avoidance behaviors. The variable-value pair representing the spawnflag is given in the spawnflag parameter in the behavior file. Multiple spawnflags may be configured for a behavior.

A runflag is meant to ”complement” an idleflag, by posting exactly when the other one does not. Similarly with the inactiveflag and activeflag. The situation is shown in Figure 17:

![Behavior Flags Diagram](image)

Figure 17: **Behavior Flags**: The four behavior flags idleflag, runflag, activeflag, and inactiveflag are posted depending on the behavior state and can be considered complementary in the manner indicated.

Behavior authors may implement their behaviors to post other messages as they see fit. For example the waypoint behavior used in the Alpha example in Section 4 also published the variable WPT_STAT with a status message similar to "vname=alpha,index=0,dist=124,eta=62" indicating the name of the vehicle, the index of the next point in the list of waypoints, the distance to that waypoint, and
the estimated time of arrival, in seconds. (You might want to re-run the Alpha mission with uXMS scoping on this variable to watch it change as the mission unfolds.)

6.5.5 Monitoring Behavior Run States and Messages During Mission Execution

The run states for each behavior, are wrapped up on each iteration by the helm into a single string and published in the variable IVPHELM_SUMMARY. This variable is subscribed for by the uHelmScope tool and behavior states are parsed from this variable and summarized in the main output of uHelmScope, as in the below excerpt:

```
12  Behaviors Active: --------- (1)
13   waypt_survey (13.0) (pwt=100.00) (pcs=1227) (cpu=0.01) (upd=0/0)
14  Behaviors Running: ------- (0)
15  Behaviors Idle: --------- (1)
16   waypt_return (22.8)
17  Behaviors Completed: ------ (0)
```

Behaviors are grouped into the four possible states, with a summary line for each state, e.g., lines 12, 14, 15, 17, containing the number of behaviors in that state in parentheses at the end of the line. Each behavior configured for the helm shows up on a dedicated line in the appropriate group, e.g., lines 13 and 16. In these lines immediately following the behavior name, the number of seconds is displayed in parentheses indicating how long the behavior has been in that state.

6.6 Behavior Reconciliation in the IvP Helm - Multi-Objective Optimization

6.6.1 IvP Functions

IvP functions are produced by behaviors to influence the decision produced by the helm on the current iteration (see Figure 13). The decision is typically comprised of the desired heading, speed, and depth but the helm decision space could be comprised of any arbitrary configuration (see Section 5.4.6). Some points about IvP functions:

- IvP functions are piecewise linearly defined. Each piece is defined by an interval over some subset of the decision space, and there is a linear function associated with each piece (see Figure 19).
- IvP functions are an approximation of an underlying function. The linear function for a single piece is the best linear approximation of the underlying function for the portion of the domain covered by that piece.
- IvP domains are discrete with an upper and lower bound for each variable, so an IvP function may achieve zero-error in approximating an underlying function by associating a piece with each point in the domain. Behaviors seldom need to do so in practice however.
- The Ivp function construct and IvP solver are generalizable to N dimensions.
• The pieces in IvP functions need not be uniform size or shape. More pieces can be dedicated to parts of the domain that are harder to approximate with linear functions.

• IvP functions need only be defined over a subset of the domain. Behaviors are not affected if the helm is configured for additional variables that a behavior may not care about. Behaviors that produce functions solely over vehicle depth are perfectly ok.

How are IvP functions built? The IvP Build Toolbox is a set of tools for creating IvP functions based on any underlying function defined over an IvP Domain. Many, if not all of the behaviors in this document make use of this toolbox, and authors of new behaviors have this at their disposal. A primary component of writing a new behavior is the development of the "underlying function", the function approximated by an IvP function with the help of the toolbox. The underlying function represents the relationship between a candidate helm decision and the expected utility with respect to the behavior’s objectives. The IvP Toolbox is not covered in detail in this document, but an overview is given below.

6.6.2 The IvP Build Toolbox

The IvP Toolbox is a set of tools (a C++ library) for building IvP functions. It is typically utilized by behavior authors in a sequence of library calls within a behavior’s (C++) implementation. There are two sets of tools - the Reflector tools for building IvP functions in N dimensions, and the ZAIC tools for building IvP functions in one dimension as a special case. The Reflector tools work by making available a function to be approximated by an IvP function. The tools simply need this function for sampling. Consider the Gaussian function rendered below in Figure 18:

![Figure 18: A rendering of the function $f(x, y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where $A = \text{range} = 150$, $\sigma = \text{sigma} = 32.4$, $x_0 = x\text{cent} = 50$, $y_0 = y\text{cent} = -150$. The domain here for $x$ and $y$ ranges from $-250$ to $250$.](image)

The ‘x’ and ‘y’ variables, each with a range of [-250, 250], are discrete, taking on integer values. The domain therefore contains $501^2 = 251,001$ points, or possible decisions. The IvP Build Toolbox
can generate an IvP function approximating this function over this domain by using a uniform piece size, as rendered in Figure 19. The only difference in these four piecewise function is the number and size of the piece. More pieces (Figure 19 (a)) results in a more accurate approximation of the underlying function, but takes longer to generate and creates further work for the IvP solver when the functions are combined. IvP functions need not use uniformly sized pieces.

Figure 19: A rendering of four different IvP functions approximating the same underlying function: The function in (a) uses a uniform distribution of 7056 pieces. The function in (b) uses a uniform distribution of 1024 pieces. The function in (c) was created by first building a uniform distribution of 49 pieces and then focusing the refinement on a sub-domain of the function. This is called directed-refinement in the IvP Build toolbox. The function in (d) was created by first building a uniform function of 25 pieces and repeatedly refining the function based on which pieces were noted to have a poor fit to the underlying function. This is termed smart-refinement in the IvP Build toolbox.

By using the directed refinement option in the IvP Build Toolbox, an initially uniform IvP function can be further refined with more pieces over a sub-domain directed by the caller, with smaller uniform pieces of the caller’s choosing. This is described more fully in the documentation for the IvP Build Toolbox. Using this tool requires the caller to have some idea where, in the sub-domain, further refinement is needed or desired. Often a behavior author indeed has this insight. For example, if one of the domain variables is vehicle heading, it may be good to have a fine refinement in the neighborhood of heading values close to the vehicle’s current heading.
In other situations, insight into where further refinement is needed may not be available to the caller. In these cases, using the smart refinement option of the IvP Build Toolbox, an initially uniform IvP function may be further refined by asking the toolbox to automatically "grade" the pieces as they are being created. The grading is in terms of how accurate the linear fit is between the piece’s linear function and the underlying function over the sub-domain for that piece. A priority queue is maintained based on the grades, and pieces where poor fits are noted, are automatically refined further, up to a maximum piece limit chosen by the caller. This is described more fully in the documentation for the IvP Build Toolbox.

The Reflector tools work similarly in N dimensions and on multi-modal functions. The only requirement for using the Reflector tool is to provide it with access to the underlying function. Since the tool repetitively samples this function, a central challenge to the user of the toolbox is to develop a fast implementation of the function. In terms of the time consumed in generating IvP functions with the Reflector tool, the sampling of the underlying function is typically the long pole in the tent.

6.6.3 The IvP Solver and Behavior Priority Weights

The IvP Solver collects a set of weighted IvP functions produced by each of the behaviors and finds a point in the decision space that optimizes the weighted combination. If each IvP objective function is represented by $f_i(x)$, and the weight of each function is given by $w_i$, the solution to a problem with $k$ functions is given by:

$$\bar{x}^* = \arg \max_{\tilde{x}} \sum_{i=0}^{k-1} w_if_i(\tilde{x})$$

The algorithm is described in detail in [29], but is summarized in the following few points.

- **The search tree**: The structure of the search algorithm is branch-and-bound. The search tree is comprised of an IvP function at each layer, and the nodes at each layer are comprised of the individual pieces from the function at that layer. A leaf node represents a single piece from each function. A node in the tree is realizable if the piece from that node and its ancestors intersect, i.e., share common points in the decision space.

- **Global optimality**: Each point in the decision space is in exactly one piece in each IvP function and is thus in exactly one leaf node of the search tree. If the search tree is expanded fully, or pruned properly (only when the pruned out sub-tree does not contain the optimal solution), then the search is guaranteed to produce the globally optimal solution. The search algorithm employed by the IvP solver does indeed start with a fully expanded tree, and utilizes proper pruning to guarantee global optimality. The algorithm does allow for a parameter for guaranteed limited back-off from the global optimality - a quicker solution with a guarantee of being within a fixed percent of global optima. This option is not exposed to the IvP Helm which always finds the global optimum.

- **Initial solution**: A key factor of an effective branch-and-bound algorithm is seeding the search with a decent initial solution. In the IvP Helm, the initial solution used is the solution (typically...
heading, speed, depth) generated on the previous helm iteration. Upon casual observation this appears to provide a speed-up by about a factor of two.

In cases where there is a "tie" between optimal decisions, the solution generated by the solver is non-deterministic. This is mitigated somewhat by the fact that the solution is seeded with the output of the previous iteration as discussed above.

### 6.6.4 Monitoring the IvP Solver During Mission Execution

The performance of the solver can be monitored with the uHelmScope tool. The output shown below is an excerpt from an example mission. On line 5, the total time needed to solve the multi-objective optimization problem is given in seconds, and the max time need for all recorded loops is given in parentheses. It is zero here since there is only one objective function in this example. On line 6 is the total time for creating the IvP functions in all behaviors, with the max across all iterations in parentheses. On line 7 is the total loop time - the sum of the previous two lines. Active behaviors display useful information regarding the IvP solver. For example, on line 13, the Survey waypoint behavior had a priority weight of 100 and generated 1,227 pieces, taking 0.01 seconds of CPU time to create.

**Listing 6.3: Example uHelmScope output containing information about the IvP solver.**

```plaintext
1 ============== uHelmScope Report ============== DRIVE (17)
2 Helm Iteration: 66  (hz=0.38)(5)  (hz=0.35)(66)  (hz=0.56)(max)
3 IvP functions: 1
4 Mode(s): Surveying
5 SolveTime:  0.00  (max=0.00)
6 CreateTime:  0.02  (max=0.02)
7 LoopTime:   0.02  (max=0.02)
8 Halted: false (0 warnings)
9 Helm Decision: [speed,0,4,21] [course,0,359,360]
10 speed = 3.00
11 course = 177.00
12 Behaviors Active: --------- (1)
13 waypt_survey (13.0) (put=100.00) (pcs=1227) (cpu=0.01) (upd=0/0)
14 Behaviors Running: --------- (0)
15 Behaviors Idle: ----------- (1)
16 waypt_return (22.8)
17 Behaviors Completed: ------ (0)
```

The solver can be additionally monitored and analyzed through the two MOOS variables LOOP_CPU and CREATE_CPU published on each helm iteration. The former indicates the system wall time for building each IvP function and solving the multi-objective optimization problem, and the latter indicates just the time to create the IvP functions.
7 Properties of Helm Behaviors

The objective of this section is to describe properties common to all IvP Helm behaviors, describe how to overload standard functions for 3rd party behaviors, and to provide a detailed simple example of a behavior. It builds on the discussion from Section 6. The focus in this section is an expansion of detail of Step 3 in Figure 13.

7.1 Brief Overview

Behaviors are implemented as C++ classes with the helm having one or more instances at runtime, each with a unique descriptor. The properties and implemented functions of a particular behavior are partly derived from the IvPBehavior superclass, shown in Figure 20. The is-a relationship of a derived class provides a form of code re-use as well as a common interface for constructing mission files with behaviors.

The IvPBehavior class provides five virtual functions which are typically overloaded in a particular behavior implementation:

- The setParam() function: parameter-value pairs are handled to configure a behavior’s unique properties distinct from its superclass.
- The onRunState() function: the meat of a behavior implementation, performed when the behavior has met its conditions for running, with the output being an objective function and a possibly empty set of variable-value pairs for posting to the MOOSDB.
- The onIdleState() function: what the behavior does when it has not met its run conditions. It may involve updating internal state history, generation of variable-value pairs for posting to the MOOSDB, or absolutely nothing at all.

![Figure 20: Behavior inheritance](image)

Figure 20: Behavior inheritance: Behaviors are derived from the IvPBehavior superclass. The native behaviors are the behaviors distributed with the helm. New behaviors also need to be a subclass of the IvPBehavior class to work with the helm. Certain virtual functions invoked by the helm may be optionally but typically overloaded in all new behaviors. Other private functions may be invoked within a behavior function as a way of facilitating common tasks involved in implementing a behavior.

The IvPBehavior class provides five virtual functions which are typically overloaded in a particular behavior implementation:
The onIdleToRunState() function: invoked once by the helm upon transitioning from the idle to running state (compared to the onRunState() function which is invoked on each helm iteration where the behavior has met its conditions).

The onRunToIdleState() function: invoked once by the helm upon transitioning from the running to idle state (compared to the onIdleState() function which is invoked on each helm iteration where the behavior has not met its conditions).

This section discusses the properties of the IvPBehavior superclass that an author of a third-party behavior needs to be aware of in implementing new behaviors. It is also relevant material for users of the native behaviors as it details general properties.

7.2 Parameters Common to All IvP Behaviors

A behavior has a standard set of parameters defined at the IvPBehavior level as well as unique parameters defined at the subclass level. By configuring a behavior during mission planning, the setting of parameters is the primary venue for affecting the overall autonomy behavior in a vehicle. Parameters are set in the behavior file, but can also be dynamically altered once the mission has commenced. A parameter is set with a single line of the form:

```
parameter = value
```

The left-hand side, the parameter component, is case insensitive, while the value component is typically case sensitive. This was discussed in depth in Section 6.3. In this section, the parameters defined at the superclass level and available to all behaviors are exhaustively listed and discussed. Each behavior typically augments these parameters with new ones unique to the behavior, and in the next section the issue of implementing new parameters by overloading the setParam() function is addressed.

7.2.1 A Summary of the Full Set of General Behavior Parameters

The following parameters are defined for all behaviors at the superclass level. They are listed here for reference - certain related aspects are discussed in further detail in other sections.

**Listing 7.1: Configuration Parameters Common to All Behaviors.**

- **name**: The name of the behavior - should be unique between all behaviors. Due to the implementation of behavior templating, spawned behavior take on a new name by concatenating on the end of a base name. For this reason all configured behavior names could be regarded as base names. The check for uniqueness includes hypothetical extension. Thus the names loiter and loiter_two are not regarded as safe since the first could potentially grow into the latter with the concatenation of _two. Logging and output sent to the helm console during operation will organize information by the behavior name.

- **priority**: The priority weight of the produced objective function. The default value is 100. A behavior may also be implemented to determine its own priority weight depending on information about the world.
duration: The time in seconds that the behavior will remain running before declaring completion. If no duration value is provided, the behavior will never time-out. The clock starts ticking once the behavior satisfies its run conditions (becoming non-idle) the first time. Should the behavior switch between running and idle states, the clock keeps ticking even during the idle periods. See Section 7.2.6 for more detail.

duration_status: If the duration parameter is set, the remaining duration time, in seconds, can be posted by naming a duration_status variable. This variable will be update/posted only when the behavior is in the running state. See Section 7.2.6 for more detail.

duration_reset: This parameter takes a variable-pair such as MY_RESET=true. If the duration parameter is set, the duration clock is reset when the variable is posted to the MOOSDB with the specified value. Each time such a post is noted, the duration clock is reset. See Section 7.2.6 for more detail.

post_mapping: This parameter takes a comma-separated pair such as WPT_STAT,WAYPT_STATUS where the left-hand value is a variable normally posted by the behavior, and the right-hand value is an alternative variable name to be used. There is no error-checking to ensure that the left-hand value names a variable actually posted by the behavior. Transitive relationships are not respected. For example, if the two re-mappings are declared, FOO,BAR, and BAR,CAR, FOO will be posted as BAR, not CAR. To disable the normal posting of a variable FOO, use post_mapping = FOO,SILENT.

duration_idle_decay: If this parameter is false the duration clock is paused when the vehicle is in the idle state. The default value is true. See Section 7.2.6 for more detail.

condition: This parameter specifies a condition that must be met for the behavior to be active. Conditions are checked for each behavior at the beginning of each control loop iteration. Conditions are based on current MOOS variables, such as STATE = normal or (K < 4). More than one condition may be provided, as a convenience, treated collectively as a single conjunctive condition. The helm automatically subscribes for any condition variables. See Section 6.5.1 for more detail on run conditions.

runflag: This parameter specifies a variable and a value to be posted when the behavior has met all its conditions for being in the running state. It is only posted if the behavior, on the previous helm iteration, was not in the running state. It is an equal-separated pair such as TRANSITING=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors.

idleflag: This parameter specifies a variable and a value to be posted when the behavior is in the idle state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was not in the idle state. It is an equal-separated pair such as WAITING=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors.
activeflag: This parameter specifies a variable and a value to be posted when the behavior is in the active state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was not in the active state. It is an equal-separated pair such as TRANSITING=true. More than one flag may be provided. These can be used to satisfy or block the conditions of other behaviors.

inactiveflag: This parameter specifies a variable and a value to be posted when the behavior is not in the active state. See the Section 6.5.3 for more on run states. It is only posted if the behavior, on the previous helm iteration, was in the active state. It is a equal-separated pair such as OUT_OF_RANGE=true. More than one flag may be specified by the user in the behavior configuration. These can be used to satisfy or block the conditions of other behaviors.

deflag: This parameter specifies a variable and a value to be posted when the behavior has set the completed state variable to be true. The circumstances causing completion are unique to the individual behavior. However, if the behavior has a duration specified, the completed flag is set to true when the duration is exceeded. The value of this parameter is a equal-separated pair such as ARRIVED_HOME=true. Once the completed flag is set to true for a behavior, it remains inactive thereafter, regardless of future events, barring a complete helm restart.

updates: This parameter specifies a variable from which updates to behavior configuration parameters are read from after the behavior has been initially instantiated and configured at the helm startup time. Any parameter and value pair that would have been legal at startup time is legal at run-time. The syntax for this string is a #-separated list of parameter-value pairs: "param=value # param=value # ... # param=value". This is one of the primary hooks to the helm for mission control - the other being the behavior conditions described above. See Section 7.2.5 for more detail.

nostarve: The nostarve parameter allows a behavior to assert a maximum staleness for one or more MOOS variables, i.e., the time since the variable was last updated. The syntax for this parameter is a comma-separated pair "variable, ..., variable, value", where last component in the list is the time value given in seconds. See Section 7.2.9 for more detail.

perpetual: Setting the perpetual parameter to true allows the behavior to continue to run even after it has completed and posted its end flags. The parameter value is not case sensitive and the only two legal values are true and false. See Section 7.2.7 for more detail.

templating: The templating parameter may be used to turn a behavior specification into a template for spawning new behaviors dynamically after the helm has been launched. Instantiation requests are received via the updates parameter described in Section 7.7.
7.2.2 The name Parameter

The name parameter is a mandatory parameter with no default value that assigns a name to the behavior instance. The name must be unique. The behavior name is case sensitive, so having the names "return_home" and "Return_Home" is acceptable (although not advisable).

Due to the capability of behavior templating (Section 7.7) the policy of behavior uniqueness was expanded to include pairs of names where one name matches the beginning component of another. For example "return" and "return_home" are not allowed. This is due to way template spawning is implemented. The spawning process begins by setting the name of the newly spawned behavior. This is achieved by appending to the base name declared for the behavior template. By allowing, for example, "return" and "return_home", it leaves open the possibility that the first behavior may spawn an instance with a name clash if it were spawned with a suffix directive of "_home".

Behavior names have no relation to the behavior class name. For example an instance of the Waypoint behavior may be called "station-keep" (although not advisable). Furthermore, multiple instances of the same behavior may certainly co-exist so long as each instance has a unique name. If the helm is configured with one or more behaviors with non-unique names, the helm will come up in the "MALCONFIG" state. The terminal or appcasting output would look something like:

```
===================================================================
pHelmTvP alpha 1/0(411)
===================================================================
Configuration Warnings: 1
[1 of 1]: Duplicate behavior name found: waypt_return

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!  !! The helm is in the MALCONFIG state due to !!
!!  !! unresolved configuration warnings.      !!
!!  !!                                    !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

7.2.3 The priority Parameter

The priority parameter is an optional parameter with a default value of 100. It assigns a priority weight to be applied to an objective function produced by the behavior. Many behaviors interpret this priority to establish a range of priorities depending on the further evaluation of the situation by the behavior. For example, the collision avoidance behavior will treat the configured priority to be the maximum priority (when a contact is close) and will degrade the priority accordingly as the contact range is greater.

The range of acceptable values is from zero to infinity. If a negative value is given for a behavior the helm will come up in the "MALCONFIG" state. The terminal or appcasting output would look something like:
7.2.4 The post_mapping Parameter

The post_mapping parameter is an optional parameter with no default value. It allows the user to configure a behavior to change the name of the variables otherwise posted by the behavior. For example, the Waypoint behavior by default publishes a variable WPT_INDEX indicating the index of the next point on its path. If there are multiple waypoint behaviors, or if you just don’t like this variable, it may be changed for example to SURVEY_INDEX with the following configuration:

```
post_mapping=WPT_INDEX,SURVEY_INDEX
```

7.2.5 Altering Parameters Dynamically with the updates Parameter

The parameters of a behavior can be made to allow dynamic modifications - after the helm has been launched and executing the initial mission in the behavior file. The modifications come in a single MOOS variable specified by the parameter updates. For example, consider the simple waypoint behavior configuration below in Listing 2. The return point is the (0,0) point in local coordinates, and return speed is 2.0 meters/second. When the conditions are met, this is what will be executed.

Listing 7.2: An example behavior configuration using the updates parameter.

```
1 Behavior = BHV_Waypoint
2 {
3   name = WAYPT_RETURN
4   priority = 100
5   speed = 2.0
6   radius = 8.0
7   points = 0,0
8   updates = RETURN_UPDATES
9   condition = RETURN = true
10  condition = DEPLOY = true
11 }
```

If, during the course of events, a different return point or speed is desired, this behavior can be altered dynamically by writing to the variable specified by the updates parameter, in this case the variable RETURN_UPDATES (line 8 in Listing 2). The syntax for this variable is of the form:

```
parameter = value # parameter = value # ... # parameter = value
```
White space is ignored. The '#' character is treated as special for parsing the line into separate parameter-value pairs. It cannot be part of a parameter component or value component. For example, the return point and speed for this behavior could be altered by any other MOOS process that writes to the MOOS variable:

```
RETURN_UPDATES = "points=50,50 # speed = 1.5"
```

Each parameter-value pair is passed to the same parameter setting routines used by the behavior on initialization. The only difference is that an erroneous parameter-value pair will simply be ignored as opposed to halting the helm as done on startup. If a faulty parameter-value pair is encountered, a warning will be written to the variable `BHV_WARNING`. For example:

```
BHV_WARNING = "Faulty update for behavior: WAYPT_RETURN. Bad parameter(s): speed."
```

Note that a check for parameter updates is made at the outset of helm iteration loop for a behavior (Figure 20) with the call `checkUpdates()`. Any updates received by the helm on the current iteration will be applied prior to behavior execution and in effect for the current iteration.

### 7.2.6 Limiting Behavior Duration with the duration Parameter

The `duration` parameter specifies a time period in seconds before a behavior times out and permanently enters the completed state (Figure 21). If left unspecified, there is no time limit to the behavior. By default, the duration clock begins ticking as soon as the helm is in drive. The duration clock remains ticking when or if the behavior subsequently enters the idle state. It even remains ticking if the helm temporarily parks. When a timeout occurs, end flags are posted. The behavior can be configured to post the time remaining before a timeout with the `duration_status` parameter. The forms for each are:

```
duration = value (positive numerical)
duration_status = value (variable name)
```

Note that the duration status variable will only be published/updated when the behavior is in the running state. The duration status is rounded to the nearest integer until less than ten seconds remain, after which the time is posted out to two decimal places. The behavior can be configured to have the duration clock pause when it is in the idle state with the following:

```
duration_idle_decay = false // The default is true
```

Configured in the above manner, a behavior’s duration clock will remain paused until its conditions are met. The behavior may also be configured to allow for the duration clock to be reset upon the writing of a MOOS variable with a particular value. For example:

```
duration_reset = BRAVO_TIMER_RESET=true
```
The behavior checks for and notes that the variable-value pair holds true and the duration clock is then reset to the original duration value. The behavior also marks the time at which the variable-value pair was noted to have held true. Thus there is no need to "un-set" the variable-value pair, e.g., setting BRAVO_TIMER_RESET=false, to allow the duration clock to resume its count-down.

7.2.7 The perpetual Parameter

When a behavior enters the completed state, it by default remains in that state with no chance to change. When the perpetual parameter is set to true, a behavior that is declared to be complete does not actually enter the complete state but performs all the other activity normally associated with completion, such as the posting of end flags. See Section 6.5.4 for more detail on posting flags to the MOOSDB from the helm. The default value for perpetual is false. The form for this parameter is:

    perpetual = value

The value component is case insensitive, and the only legal values are either true or false. A behavior using the duration parameter with perpetual set to true will post its end flags upon time out, but will reset its clock and begin the count-down once more the next time its run conditions are met, i.e., enters the running state. Typically when a behavior is used in this way, it also posts an endflag that would put itself in the idle state, waiting for an external event.

7.2.8 The post_mapping Parameter

This parameter takes a comma-separated pair where the left-hand value is a variable normally posted by the behavior, and the right-hand value is an alternative variable name to be used. For example, the Waypoint behavior normally posts the variable WPT_STAT when it is running. This can be changed with:

    post_mapping = WPT_STAT, WAYPT_STATUS

Note there is no error-checking to ensure that the left-hand value names a variable actually posted by the behavior. Transitive relationships are not respected. For example, if the two re-mappings are declared, FOO,BAR, and BAR,CAR, FOO will be posted as BAR, not CAR. To disable the normal posting of a variable FOO, use:

    post_mapping = FOO, SILENT

7.2.9 Detection of Stale Variables with the nostarve Parameter

A behavior utilizing a variable generated by a MOOS process outside the helm, may require the variable to be sufficiently up-to-date. The staleness of a variable is the time since it was last written to by any process. The nostarve parameter allows the mission writer to set a staleness threshold. The form for this parameters is:
nostarve = variable_1, ..., variable_n, duration

The value of this parameter is a comma-separated list such as nostarve = NAV_X, NAV_Y, 5.0. The variable components name MOOS variables and the duration component, the last entry in the list, represents the tolerated staleness in seconds. If staleness is detected, a behavior failure condition is triggered which will trigger the helm to post all-stop values and relinquish to manual control.

7.3 Overloading the setParam() Function in New Behaviors

The setParam() function is a virtual function defined in the IvPBehavior class, with parameters implemented in the superclass (Section 7.2) handled in the superclass version of this function:

    bool IvPBehavior::setParam(string parameter, string value);

The setParam() function should return true if the parameter is recognized and the value is in an acceptable form. In the rare case that a new behavior has no additional parameters, leaving this function undefined in the subclass is appropriate. The example below in Listing 3 gives an example for a fictional behavior BHV_YourBehavior having a single parameter period.

Listing 7.3: Example setParam() implementation for fictional BHV_YourBehavior.

```cpp
1   bool BHV_YourBehavior::setParam(string param, string value)  
2   {  
3       if(param == "period") {  
4           double time_value = atof(value.c_str());  
5           if((time_value < 0) || (!isNumber(value)))  
6               return(false);  
7           m_period = time_value;  
8           return(true);  
9       }  
10   }  
11   return(false);  
```

Since the period parameter refers to a time period, a check is made on line 4 that the value component indeed is not a negative number. The atof() function on line 6, which converts an ASCII string to a floating point value, returns zero when passed a non-numerical string, therefore the isNumber() function is also used to ensure the string represented by value represents a numerical value. The isNumber() function is part of the IvP utility library. A behavior implementation of this function without sufficient syntax or semantic checking simply runs the risk that faulty parameters are not detected at the time of helm launch, or during dynamic updates. Solid checking in this function will reduce debugging headaches down the road.

7.4 Behavior Functions Invoked by the Helm

The IvPBehavior superclass implements a number of functions invoked by the helm on each iteration. Two of these functions are overloadable as described previously - the onRunState() and onIdleState() functions. The basic flow of calls to a behavior from the helm are shown in Figure 21. These are
discussed in more detail later in the section, but the idea is to execute certain behavior functions based on the *activity state*, which may be one of the four states depicted.

Figure 21: **Behavior function-calls by the helm:** The helm invokes a sequence of functions on each behavior on each iteration of the helm. The sequence of calls is dependent on what the behavior returns, and reflects the behaviors activity state. Certain functions are immutable and cannot be overloaded by a behavior author. Two key functions, **onRunState()** and **onIdleState()** can be indeed overloaded as the usual hook for an author to provide the implementation of a behavior. The **postFlags()** function is also immutable, but the parameters (flags) are provided in the helm configuration (*.bhv) file.

An *idle* behavior is one that has not met its conditions for running. A *completed* behavior is one that has reached its objectives or exceeded its duration. A *running* behavior is one that has not yet completed, has met its run conditions, but may still opt not to produce any output. An *active* behavior is one that is running and is producing output in the form of an objective function.

The types of functions defined at the superclass level fall into one of the three categories below, only the first two of which are shown in Figure 21:

- Helm-invoked immutable functions - functions invoked by the helm on each iteration that the author of a new behavior may not re-implement.
- Helm-invoked overloadable functions - functions invoked by the helm that an author of a new behavior typically re-implements of overloads.
User-invoked functions - functions invoked within a behavior implementation.

The user-invoked functions are utilities for common operations typically invoked within the implementation of the `onRunState()` and `onIdleState()` functions written by the behavior author.

### 7.4.1 Helm-Invoked Immutable Functions

These functions, implemented in the `IvPBehavior` superclass, are called by the helm but are not defined as virtual functions which means that attempts to overload them in a new behavior implementation will be ignored. See Figure 21 regarding the sequence of these function calls.

- **void checkUpdates():** This function is called first on each iteration to handle requested dynamic changes in the behavior configuration. This needs to be the very first function applied to a behavior on the helm iteration so any requested changes to the behavior parameters may be applied on the present iteration. See Section 7.2.5 for more on dynamic behavior configuration with the `updates` parameter.

- **bool isComplete():** This function simply returns a Boolean indicating whether the behavior was put into the `complete` state during a prior iteration.

- **bool isRunnable():** Determines if a behavior is in the `running` state or not. Within this function call four things are checked: (a) if the `duration` is set, the duration time remaining is checked for timeout, (b) variables that are monitored for staleness are checked against (Section 7.2.9), (c) the run conditions must be met. (d) the behavior’s decision domain (IvP domain) is a proper subset of the helm’s configured IvP domain. See Section 6.5.1 for more detail on run conditions.

- **void postFlags(string flag_type):** This function will post flags depending on whether the value of `flag_type` is set to "idleflags", "runflags", "activeflags", "inactiveflags", or "endflags". Although this function is immutable, not overloadable by subclass implementations, its effect is indeed mutable since the flags are specified in the mission configuration `*.bhv` file. See Section 6.5.4 for more detail on posting flags to the MOOSDB from the helm.

### 7.4.2 Helm-Invoked Overloaded Functions

These are functions called by the helm. They are defined as virtual functions so that a behavior author may overload them. Typically the bulk of writing a new behavior resides in implementing these three functions.

- **IvPFunction* onRunState():** This function is called by the helm when deemed to be in the `running` state (Figure 21). The bulk of the work in implementing a new behavior is in this function implementation, and is the subject of Section 7.6.

- **void onIdleState():** This function is called by the helm when deemed to be in the `idle` state (Figure 21). Many behaviors are implemented with this function left undefined, but it is a useful hook to have in many cases.
- **bool setParam(string, string)**: This function is called by the helm when the behavior is first instantiated with the set of parameter and parameter values provided in the behavior file. It is also called by the helm within the `checkUpdates()` function to apply parameter updates dynamically.

#### 7.5 Local Behavior Utility Functions

The bulk of the work done in implementing a new behavior is in the implementation of the `onIdleState()` and `onRunState()` functions. The utility functions described below are designed to aid in that implementation and are generally "protected" functions, that is callable only from within the code of another function in the behavior, such as the `onRunState()` and `onIdleState()` functions, and not invoked by the helm.

#### 7.5.1 Summary of Implementor-Invoked Utility Functions

The following is summary of utility functions implemented at the `IvPBehavior` superclass level.

- **void setComplete()**: The notion of what it means for a behavior to be "complete" is largely an issue specific to an individual behavior. When or if this state is reached, a call to `setComplete()` can be made and end flags will be posted, and the behavior will be permanently put into the `completed` state unless the `perpetual` parameter is set to true. See Section 6.5.3 for more on behavior run states.

- **void addInfoVars(string var_names)**: The helm will register for variables from the MOOSDB on a need-only basis, and a behavior is obligated to inform the helm that certain variables are needed on its behalf. A call to the `addInfoVars()` function can be made from anywhere with a behavior implementation to declare needed variables. This can be one call per variable, or the string argument can be a comma-separated list of variables. The most common point of invoking this function is within a behavior’s constructor since needed variables are typically known at the point of instantiation. More on this in Section 7.5.3.

- **double getBufferDoubleVal(string varname, bool& result)**: Query the info_buffer for the latest (double) value for a given variable named by the string argument. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

- **double getBufferStringVal(string varname, bool& result)**: Query the info_buffer for the latest (string) value for a given variable named by the string argument. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

- **double getBufferCurrTime()**: Query the info_buffer for the current buffer local time, equivalent to the duration in seconds since the helm was launched. More on this in Section 7.5.2.

- **vector<double> getBufferDoubleVector(string var, bool& result)**: Query the info_buffer for all changes to the variable (of type double) named by the string argument, since the last iteration. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.
• vector<string> getBufferStringVector(string var, bool& result): Query the info_buffer for all changes to the variable (of type string) named by the string argument, since the last iteration. The bool argument indicates whether the queried variable was found in the buffer. More on this in Section 7.5.2.

• void postMessage(string varname, string value, string key): The helm can post messages (variable-value pairs) to the MOOSDB at the end of the helm iteration. Behaviors can request such postings via a call to the postMessage() function where the first argument is the variable name, and the second is the variable value. The optional key parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

• void postMessage(string varname, double value, string key): Same as above except used when the posted variable is of type double rather than string. The optional key parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

• void postBoolMessage(string varname, bool value, string key): Same as above, except used when the posted variable is a bool rather than string. The optional key parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

• void postIntMessage(string varname, double value, string key): Same as postMessage(string, double) above except the numerical output is rounded to the nearest integer. This, combined with the helm’s use of the duplication filter, can reduce the number of posts to the MOOSDB. The optional key parameter is used in conjunction with the duplication filter and by default is the empty string. See Section 5.8 for more on the duplication filter.

• void postWMessage(string warning_msg): Identical to the postMessage() function except the variable name is automatically set to BHV_WARNING. Provided as a matter of convenience to the caller and for uniformity in monitoring warnings.

• void postEMessage(string error_msg): Similar to the postWMessage() function except the variable name is BHV_ERROR. This call is for more serious problems noted by the behavior. It also results in an internal state_ok bit being flipped which results in the helm posting all-stop values to the actuators.

• void postRepeatableMessage(string varname, string value): A convenience function. A posting of postRepeatableMessage("FOO", "bar") is equivalent to postMessage("FOO", "bar", "repeatable").

• void postRepeatableMessage(string varname, double value): A convenience function. A posting of postRepeatableMessage("FOO", 100) is equivalent to postMessage("FOO", 100, "repeatable").

7.5.2 The Information Buffer
Behaviors do not have direct access to the MOOSDB - they don’t read mail, and they don’t post changes directly, but rather through the helm as an intermediary. The information buffer, or info_buffer, is
a data structure maintained by the helm to reflect a subset of the information in the MOOSDB and made available to each behavior. This topic is hidden from a user configuring existing behaviors and can be safely skipped, but is an important issue for a behavior author implementing a new behavior. The info_buffer is a data structure shared by all behaviors, each behavior having an pointer to a single instance of the InfoBuffer class. This data structure is maintained by the helm, primarily by reading mail from the MOOSDB and reflecting the change onto the buffer on each helm iteration, before the helm requests input from each behavior. Each behavior therefore has the exact same snapshot of a subset of the MOOSDB. A behavior author needs to know two things - how to ensure that certain variables show up in the buffer, and how to access that information from within the behavior. These two issues are discussed next.

7.5.3 Requesting the Inclusion of a Variable in the Information Buffer

A variable can be specifically requested for inclusion in the info_buffer by invoking the following function:

```c++
void IvPBehavior::addInfoVars(string varnames)
```

The string argument is either a single MOOS variable or a comma-separated list of variables. Duplicate requests are simply ignored. Typically such calls are invoked in a behavior’s constructor, but may be done dynamically at any point after the helm is running. The helm will simply register with the MOOSDB for the requested variable at the end of the current iteration. Certain variables are registered for automatically on behalf of the behavior. All variables referenced in run conditions will be registered and accessible in the buffer. Variables named in the updates and nostarve parameters will also be automatically registered.

7.5.4 Accessing Variable Information from the Information Buffer

A variable value can be queried from the buffer with one of the following two function calls, depending on whether the variable is of type double or string.

```c++
string IvPBehavior::getBufferStringVal(string varname, bool& result)
double IvPBehavior::getBufferDoubleVal(string varname, bool& result)
```

The first string argument is the variable name, and the second argument is a reference to a Boolean variable which, upon the function return, will indicate whether the queried variable was found in the buffer. A duration value indicating the elapsed time since the variable was last changed in the buffer can be obtained from the following function call:

```c++
double IvPBehavior::getBufferTimeVal(string varname);
```

The string argument is the variable name. The returned value should be exactly zero if this variable was updated by new mail received by the helm at the beginning of the current iteration. If the variable name is not found in the buffer, the return value is -1. The “current” buffer time, equivalent to the cumulative time in seconds since the helm was launched, can be retrieved with the following function call:

```c++
double IvPBehavior::getBufferCurrTime();
```
The buffer time is a local variable of the info_buffer data structure. It is updated once at the beginning of the helm OnNewMail() loop prior to processing all new updates to the buffer from the MOOS mail stack, or at the beginning of the Iterate() loop if no mail is processed on the current iteration. Thus the time-stamp returned by the above call should be exactly the same for successive calls by all behaviors within a helm iteration.

The values returned by getBufferStringVal() and getBufferDoubleVal() represent the latest value of the variable in the MOOSDB at the point in time when the helm began its iteration and processed its mail stack. The value may have changed several times in the MOOSDB between iterations, and this information may be of use to a behavior. This is particularly true when a variable is being posted in pieces, or a sequence of delta changes to a data structure. In any event, this information can be recovered with the following two function calls:

```cpp
vector<string> IvPBehavior::getBufferStringVector(string varname, bool& result)
vector<double> IvPBehavior::getBufferDoubleVector(string varname, bool& result)
```

They return all values updated to the buffer for a given variable since the last iteration in a vector of strings or doubles respectively. The latest change is located at the highest index of the vector. An empty vector is returned if no changes were received at the outset of the current iteration.

### 7.6 Overloading the onRunState() and onIdleState() Functions

The onRunState() function is declared as a virtual function in the IvPBehavior superclass intended to be overloaded by the behavior author to accomplish the primary work of the behavior. The primary behavior output is the objective function. This is what drives the vehicle. The objective function is an instance of the class IvPFunction, and a behavior generates an instance and returns a pointer to the object in the following function:

```cpp
IvPFunction* onRunState()
```

This function is called automatically by the helm on the current iteration if the behavior is deemed to be in the running state, as depicted in Figure 21. The invocation of onRunState() does not necessarily mean an objective function is returned. The behavior may opt not to for whatever reason, in which case it returns a null pointer. However, if it does generate a function, the behavior is said to be in the active state. The steps comprising the typical implementation of the onRunState() implementation can be summarized as follows:

- Get information from the info_buffer, and update any internal behavior state.
- Generate any messages to be posted to the MOOSDB.
- Produce an objective function if warranted.
- Return.

The same steps hold for the onIdleState() function except for producing an objective function. The first two steps have been discussed in detail. Accessing the info_buffer was described in Sections 7.5.2 - 7.5.4. The functions for posting messages to the MOOSDB from within a behavior were discussed in Section 7.5.1. Further issues regarding the posting of messages were covered in Section 6.5.4 The remaining issue to discuss is how objective functions are generated. This is covered in the IvPBuild Toolbox documentation.
7.7 Dynamic Behavior Spawning

In certain scenarios it may not be practical or possible to know in advance all the behaviors needed to accomplish mission objectives. For example, if the helm uses a certain kind of behavior to deal with another vehicle in its operation area, for collision avoidance or trailing etc., the identities or number of such vehicles may not be known when the mission planner is configuring the helm’s behavior file. One way to circumvent this problem is to design a collision avoidance behavior, for example, to handle all known contacts. However, this has a couple drawbacks. It would entail a degree of multi-objective optimization be implemented within the behavior to produce an objective function that was comprehensive across all contacts. This would likely be much more computationally expensive than simply generating an objective function for each contact. It also may be advantageous to have different types of collision avoidance behaviors for different contact types or collision avoidance protocols. In any event, the helm support for dynamic behavior spawning gives behavior architects and mission planners another potentially powerful option for implementing an autonomy system.

7.7.1 Behavior Specifications Viewed as Templates

The templating parameter may be used to turn an otherwise static behavior specification into a template for spawning new behaviors dynamically after the helm has been launched. Instantiation requests are received via the updates parameter described in Section 7.2.5. Updates received through this variable are normally used to change behavior parameters dynamically, but they can be further used to request the spawning of a new behavior by including the following component:

\[
name = <new-behavior-name>
\]

If the \texttt{<new-behavior-name>\texttt{}} is not the name of the behavior given in the behavior specification, and if it is not the name of a behavior already presently instantiated by the helm, the helm interprets this as a request to spawn a new behavior, if templating is enabled. Templating is enabled by including the following component in the behavior specification:

\[
templating = <templating-mode>
\]

The \texttt{<templating-mode>\texttt{}} may be set to either "disallowed" (the default), "clone", or "spawn". In the "clone" mode, the helm will instantiate a behavior immediately upon helm startup. In the "spawn" mode, the helm will not instantiate a behavior until it receives a request to do so via the \texttt{updates\texttt{}} parameter as described above. An example of a behavior configured to allow dynamic spawning is given in Listing \texttt{1}, taken from the Berta example mission.

For a behavior configured with templating enabled in the "spawn" mode, the helm will \textit{not} spawn a behavior at the helm startup time. However, internally it will indeed spawn such a behavior, check that it can be found and built as configured, and then destroy it immediately. This means that the behavior configuration found in the \texttt{.bhv\texttt{}} configuration file must not have an invalid configuration. It is preferable to know at helm launch time that a behavior is misconfigured, rather than waiting for the spawning event to occur perhaps hours into a mission and being surprised that a critical behavior, such as collision avoidance, failed to be spawned.
7.7.2 Behavior Completion and Removal from the Helm

All behaviors, whether statically spawned upon helm startup, or dynamically spawned during the mission, are capable of dying and being removed from the helm. Death and removal are part of the consequences of a behavior entering the completed state. Behavior run states were discussed in Section 6.5.3. A completed behavior configured with `perpetual=true` will not die upon completion. Once a behavior dies, its name is removed from the helm’s internal registry of currently-spawned behaviors and a new behavior by the same name may be spawned at a future time.

7.7.3 Example Missions with Dynamic Behavior Spawning

Two example missions are provided that demonstrate the workings of dynamic behavior spawning, The Echo mission in Section 32, and the Berta mission in Section 34. The Echo mission involves a single vehicle with its helm configured to spawn dynamic behaviors of the type `BHV_BearingLine`. These behaviors do nothing more than post a viewable line segment to the MOOSDB between ownship and a point in the operation area. The interesting thing about this example is that the mission is configured with an event script (via `uTimerScript`) to automatically cue the spawning of 5000 behaviors over about one hour. Each behavior has a random duration of less than a minute, so behaviors are spawning and dying quite rapidly with visual confirmation via the viewable line segments.

The second example mission, the Berta mission, involves two vehicles that are loitering near one another. Periodically their loiter assignments are randomly altered (again through `uTimerScript`). The change in loiter locations repeatedly puts them on an unpredictable and random near-collision course and each vehicle needs to spawn a collision avoidance behavior. The interesting thing about this scenario is that the behavior, the `BHV_AvoidCollision` behavior, is an actual behavior of common use, unlike the `BHV_BearingLine` behavior used in the Echo mission. This example also uses the `pBasicContactMgr` to coordinate the receiving of contact reports with helm behavior spawning.

7.7.4 Examining the Helm’s Life Event History

Behavior spawning, and behavior completion and removal from the helm, are two types of life events the helm takes note of and posts in the MOOS variable `IVPHELM_LIFE_EVENT`. A third type of life event occurs when a behavior spawning is aborted due to either a syntax error or a name collision. Monitoring life events at run time is possible by scoping on the variable `IVPHELM_LIFE_EVENT` with either `uXMS` or `uMS`. A better method is available via the `uHelmScope` application. It automatically registers for the `IVPHELM_LIFE_EVENT` variable and will generate a formatted report like that shown in Listing 4. In the post-mission analysis phase, the `aloghelm` application may be used to examine the life event history and will generate the same formatted report from a given alog file.

Listing 7.4: A Life Event History generated with either the `uHelmScope` or `aloghelm` utilities.

```
1 ***************************************************
2 * Summary of Behavior Life Events                  *
3 ***************************************************
4 Time Iter Event  Behavior       Behavior Type Spawning Seed
5 ------ ---- ----- ------------ --------------- ----------------------------
6 47.84 1 spawn loiter BHV_Loiter helm startup
```
The life event history shown in Listing 4 was taken from the Berta example mission, the "gilda" vehicle, described in Section 34. The time-stamp reported in column one is the elapsed time between the event and the time of the helm’s startup. The first three events, in lines 6-8, reflect the three static behaviors spawned when the helm was launched, in first iteration of the helm. The collision avoidance behavior was spawned each time (lines 9, 11, 13 etc.) the vehicle "henry" came within sufficiently close range. Each time the "henry" vehicle passed and opened range to a sufficient amount, the collision avoidance behavior completed and died (lines 10, 12, 14, etc.). Line 17 shows an example of an aborted spawning. This was brought about purposely by poking the MOOSDB with the Spawning Seed shown for that line. Since the collision avoidance parameter does not have a parameter "foo", the spawning failed.

Accessing the life event history via uHelmScope may be done by launching the scope with the vehicle’s mission file, and hitting the ’L’ key to toggle into the life event history mode. Or one may launch the scope directly into this mode via:

```
uHelmScope --life targ_gilda.moos
```

The same summary may also be accessed after mission completion via the log files:

```
aloghelm --life gilda.alog
```

Note that perhaps not all life events will be displayed when using uHelmScope, depending on when it is launched relative to pHelmIvP. When uHelmScope connects to the MOOSDB it will only receive the latest and all following posts to the variable IVPHELM_LIFE_EVENT. If uHelmScope connects after pHelmIvP is launched and put into drive, it may have missed older postings. The initial spawning events do not occur in the helm until the helm enters the DRIVE state. (See Section 5.2 about helm state). In the example in Listing 4, the helm apparently was in the PARK state for about 48 seconds before it was put into drive and began to execute its first iteration. The full event history should always be accessible via the log file however.
8 Behaviors of the IvP Helm

The following is a description of some single-vehicle behaviors currently written for the helm. The below is a guide for users of these behaviors. The topic of developing new behaviors is addressed separately. The setting of behavior parameters is the primary method for affecting the overall autonomy behavior in a vehicle. Parameters may also be dynamically altered once the mission has commenced. A parameter is set with a single line of the form:

\[
\text{parameter} = \text{value}
\]

The left-hand side, the parameter component, is case insensitive, while the value component is typically case sensitive. When the helm is launched, each behavior is created and the parameters are set. If a parameter setting in the behavior file references an unknown parameter, or if the value component fails a syntactic or semantic test, the line is noted and the helm ceases to launch.

Overview of Parameters Common to All Behaviors

The parameters below are common to all IvP behaviors, although they may have more relevance to some behaviors than others. They are defined in the IvPBehavior superclass of which all the behaviors described in this section are a subclass. More information on the functionality behind these parameters was given in Section 7.2.1.

Listing 8.1: Configuration Parameters Common to All IvP Behaviors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activeflag</td>
<td>A flag (MOOSVar,Data) pair, posted when in the active state.</td>
</tr>
<tr>
<td>condition</td>
<td>A logical condition that must satisfied for the behavior to run.</td>
</tr>
<tr>
<td>duration</td>
<td>Behavior duration, in seconds. Default is &quot;unlimited&quot;.</td>
</tr>
<tr>
<td>duration_idle_decay</td>
<td>If true, clock will progress even if in the idle state.</td>
</tr>
<tr>
<td>duration_reset</td>
<td>If limited duration, duration is reset when this variable is received.</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOS variable informing of remaining duration, if duration is set.</td>
</tr>
<tr>
<td>endflag</td>
<td>A flag (MOOSVar,Data) pair, posted when the behavior completes.</td>
</tr>
<tr>
<td>idleflag</td>
<td>A flag (MOOSVar,Data) pair, posted when in the idle state.</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>A flag (MOOSVar,Data) pair, posted when in the inactive state.</td>
</tr>
<tr>
<td>name</td>
<td>A unique identifier for the behavior instance.</td>
</tr>
<tr>
<td>nostarve</td>
<td>If a given MOOSVar is stale by a given amount, an error is posted.</td>
</tr>
<tr>
<td>perpetual</td>
<td>If true, a behavior may be reset after completion or duration timeout.</td>
</tr>
<tr>
<td>post_mapping</td>
<td>A mapping to change the default MOOS variable output of a behavior.</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>Priority weight of the IvP function produced by behavior.</td>
</tr>
<tr>
<td>runflag</td>
<td>A flag (MOOSVar,Data) pair, posted when in the running state.</td>
</tr>
<tr>
<td>templating</td>
<td>Enable the behavior spec as a template for dynamic spawning.</td>
</tr>
<tr>
<td>updates</td>
<td>A MOOSVar from which behavior parameter updates are received.</td>
</tr>
</tbody>
</table>

Configuring One Variable Objective Functions

Several behaviors use a common tool for constructing objective functions over a single decision variable. These behaviors have a similar interface for configuring this tool, and it is described
here to avoid redundancy. Examples of behaviors that use this tool are the Waypoint, Loiter, PeriodicSurface, ConstantDepth, ConstantSpeed, ConstantHeading, and Shadow behaviors. This tool is called the ZAIC_PEAK tool, and is describe in more detail in [33]. This tool is designed with the objective function form shown in Figure 22 in mind, where there is an identifiable preferred single decision choice (the summit) with maximum utility, and then a gradual drop in utility as the variable varies from the preferred choice.

Figure 22: The ZAIC_PEAK tool: defines an IvP function over one variable defined by the four parameters shown here. In the case rendered here, the tool would create an IvP function with six pieces. The function rendered was created with summit=180, peakwidth=85, basewidth=70, summitdelta=40.

The form in which the utility drops is dependent on the settings of other parameters shown in the figure. The summit, peakwidth, and basewidth values are given in units native to the decision variable, while the summitdelta, minutil, and maxutil values are given in terms of units of utility. The latter two variables default to 0 and 100 respectively and are not typically exposed as configuration parameters in behaviors that use this tool, unlike the other four parameters.
9 The Waypoint Behavior

The Waypoint behavior is used for transiting to a set of specified waypoint in the x-y plane. The primary parameter is the set of waypoints. Other key parameters are the inner and outer radius around each waypoint that determine what it means to have met the conditions for moving on to the next waypoint. The basic idea is shown in Figure 23.

![Figure 23: The Waypoint behavior](image)

The behavior may also be configured to perform a degree of track-line following, that is, steering the vehicle not necessarily toward the next waypoint, but to a point on the line between the previous and next waypoint. This is to ensure the vehicle stays closer to this line in the face of external forces such as wind or current. The behavior may also be set to “repeat” the set of waypoints indefinitely, or a fixed number of times. The waypoints may be specified either directly at start-up, or supplied dynamically during operation of the vehicle. There are also a number of accepted geometry patterns that may be given in lieu of specific waypoints, such as polygons, lawnmower pattern and so on.

9.1 Configuration Parameters

Listing 9.1: Configuration Parameters Common to All Behaviors.

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the idle state. Section 7.2.6.
**duration_reset:** A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. See Section 7.2.6.

**duration_status:** The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.

**endflag:** A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.

**idleflag:** A MOOS variable-value pair posted when the behavior is in the `idle` state. Section 6.5.4.

**inactiveflag:** A MOOS variable-value posted when the behavior is `not` in the `active` state. Section 6.5.4.

**name:** The (unique) name of the behavior. Section 7.2.2.

**nostarve:** Allows a behavior to assert a maximum staleness for a MOOS variable. Section 7.2.9.

**perpetual:** If true allows the behavior to run even after it has completed. Section 7.2.7.

**post_mapping:** Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.

**priority:** The priority weight of the behavior. Section 7.2.3.

**pwt:** Same as `priority`.

**runflag:** A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.

**spawnflag:** A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

**templating:** Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

**updates:** A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

---

**Listing 9.2: Configuration Parameters for the Waypoint Behavior.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capture_radius</td>
<td>The radius tolerance, in meters, for satisfying the arrival at a waypoint. The default is 3. Section 9.5.</td>
</tr>
<tr>
<td>capture_line</td>
<td>If set to true, waypoint arrival will be achieved if the vehicle crosses the line perpendicular to the line approaching the waypoint. Default is false. Section 9.6.</td>
</tr>
<tr>
<td>crs_spd_zaic_ratio</td>
<td>Specifies the relative weight of the course portion of the ZAIC. Valid range is [1, 99]. Default is 50.</td>
</tr>
<tr>
<td>cycleflag</td>
<td>Optional MOOS variable-value pairs posted at end of each cycle through waypoints.</td>
</tr>
<tr>
<td>efficiency_measure</td>
<td>Determines if leg by leg efficiency is to be measured and/or reported. The default is &quot;off&quot;.</td>
</tr>
<tr>
<td>lead</td>
<td>If this parameter is set, track-line following between waypoints is enabled. Section 9.7.</td>
</tr>
<tr>
<td>lead_damper</td>
<td>Distance from trackline within which the <code>lead</code> distance is stretched out. Section 9.7.</td>
</tr>
</tbody>
</table>
lead_to_start: Boolean indicating whether trackline following is applied to first waypoint. The default is false. Section 9.7.

order: The order in which waypoints are traversed - "normal", "reverse", and with 14.3 or later, "toggle". Section 9.1.

points: A colon separated list of x,y pairs given as points in 2D space, in meters. Section 9.1.

point: A single x,y pair given as a point in 2D space, in meters. Section 9.1.


post_suffix: A suffix tagged onto the WPT_STATUS, WPT_INDEX and CYCLE_INDEX variables.

radius: An alias for capture_radius. Section 9.5.

repeat: The number of extra times traversed through the waypoints. Or "forever". The default is "normal" meaning the points will be visited in the order listed. Section 9.4.

slip_radius: An "outer" capture radius. Arrival declared when the vehicle is in this range and the distance to the next waypoint begins to increase. The default is 15 meters. Section 9.5.

speed: The desired speed (m/s) at which the vehicle travels through the points. Accepts only non-negative numbers. The default is 0.

speed_alt: An alternative desired speed (m/s) at which the vehicle travels through the points. Applies only when the use_alt_speed parameter is set to true. The default is -1, which indicates internally that it has not been set. (Introduced after release 15.5.)

use_alt_speed: If true then attempt to use the alternate speed set with the speed_alt parameter, if that speed is set to a non-negative value. The default is false. (Introduced after release 15.5.)

visual_hints: Optional hints for visual properties in variables posted intended for rendering. Section 9.12.

wpt_status_var: Optional MOOS variable for posting the waypoint status messages as described in Section 9.9.

wpt_index_var: Optional MOOS variable for posting the waypoint index messages as described in Section 9.9.

wptflag: Optional MOOS variable-value pairs posted after each waypoint has been reached.

Listing 9.3: Example Configuration Block.
Behavior = BHV_Waypoint
{
    // General Behavior Parameters
    // ---------------------------
    name = transit // example
    pwt = 100 // default
    condition = MODE==TRANSITING // example
    updates = TRANSIT_UPDATES // example

    // Parameters specific to this behavior
    // ------------------------------------
    capture_radius = 3 // default
    capture_line = false // default
    cycle_flag = COMMS_NEEDED=true // example
    lead = -1 // default
    lead_damper = -1 // default
    lead_to_start = false // default
    order = normal // default
    points = pts={-200,-200:30,-60} // example
    post_suffix = HENRY // example
    repeat = 0 // default
    slip_radius = 15 // default
    speed = 1.2 // default is zero
    wptflag = HITPTS = $(X),$(Y) // example

    visual_hints = vertex_size = 3 // default
    visual_hints = edge_size = 1 // default
    visual_hints = vertex_color = dodger_blue // default
    visual_hints = edge_color = white // default
    visual_hints = nextpt_color = yellow // default
    visual_hints = nextpt_lcolor = aqua // default
    visual_hints = nextpt_vertex_size = 5 // default
}

9.2 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition
to any configured flags. A variable published by any behavior may be supressed or changed to a
different variable name using the post mapping configuration parameter described in Section 7.2.8.

- WPT_STAT: A comma-separated string showing the status in hitting the list of points.
- WPT_INDEX: The index of the current waypoint. First point has index 0.
- CYCLE_INDEX: The number of times the full set of points has been traversed, if repeating.
- VIEW_POINT: A visual cue for indicating the waypoint currently heading toward.
- VIEW_INDEX: A visual cue for indicating the steering point, if the lead parameter is used.
- VIEW_SEGLIST: A visual cue for rendering the full set of waypoints.

The Waypoint behavior will also publish any MOOS variables configured with the general behavior
flags:

- runflag, idleflag, activeflag, inactiveflag, and endflag, described in Section 7.2.
as well as the flags defined locally for the Waypoint behavior:

- cyleflag and wptflag.

### 9.3 Specifying Waypoints with the points or point Parameter

There are a few formats supported for setting the list of waypoints. In the simplest case, when there is just a single point, as with the waypoint return behavior in the alpha mission, the following format may be used:

```python
point = 0,0
```

Using the plural `points` parameter is also ok when there is only one point. Another common method is to specify a colon-separated list of comma-separate pairs. For example, also from the alpha mission:

```python
points = 60,-40:60,-160:150,-160:180,-100:150,-40
```

Other formats supported are tied to the `XYSegList` and `XYPolygon` classes. The former represents a set of vertices connected by line segments with a clear first and last vertex. The latter represents a convex polygon over vertices with the notion of the first or last vertex less clear in some cases. So for example, both of the following ways of setting waypoints are supported:

```python
points = pts={60,-40:60,-160:150,-160:180,-100:150,-40}
points = format=lawnmower, x=0, y=40, height=60, width=180, lane_width=15
```

When loading a waypoint specification, the behavior will first attempt to build an `XYSegList` from one of the above formats. Failing this, it will try to build an `XYPolygon` from the specification. Therefore any of the `XYPolygon` string formats described in the Geometry documentation is supported. For example:

```python
points = format=radial, label=foxtrot, x=0, y=40, radius=60, pts=6, snap=1
points = format=ellipse, label=golf, x=0, y=40, degs=45, pts=14, snap=1, major=100, minor=70
```

In specifying a list of points indirectly via the lawnmower, radial or ellipse patterns, the "first" point in the list may be less predictable.

### 9.4 The order and repeat Parameters

The order of the parameters may be reversed with the `order` parameter, and the number of times the waypoints are traversed may be adjusted with the `repeat` parameter. An example specification:

```python
points = 60,-40:60,-160:150,-160:180,-100:150,-40
order = reverse  // default is "normal"
repeat = 3       // default is 0
```
A waypoint behavior with this specification will traverse the five points in reverse order (150, -40 first) four times (one initial cycle and then repeated three times) before completing. If there is a syntactic error in this specification at helm start-up, an output error will be generated and the helm will launch in the MALCONFIG mode. If the syntactic error is passed as part of a dynamic update (see Section 7.2.5), the change in waypoints will be ignored and the a warning posted to the BHV.WARNING variable. See the Geometry documentation for more methods for specifying sets of waypoints. The behavior can be set to repeat its waypoints indefinitely by setting repeat="forever". The point parameter may be used instead of points when there is only a single waypoint.

9.5 The capture_radius and slip_radius Parameters

The capture_radius parameter specifies the distance to a given waypoint the vehicle must be before it is considered to have arrived at or achieved that waypoint. It is the inner radius around the points in Figure 23. The slip_radius parameter specifies an alternative criteria for achieving a waypoint.

![Figure 24: The capture radius and slip radius](image)

As the vehicle progresses toward a waypoint, the sequence of measured distances to the waypoint decreases monotonically. The sequence becomes non-monotonic when it hits its waypoint or when there is a near-miss of the waypoint capture radius. The slip_radius, is a capture radius distance within which a detection of increasing distances to the waypoint is interpreted as a waypoint arrival. This distance would have to be larger than the capture radius to have any effect. As a rule of thumb, a distance of twice the capture radius is practical. The idea is shown in Figure 24. The behavior keeps a running tally of hits achieved with the capture radius and those achieved with the slip radius. These tallies are reported in a status message described in Section 9.9 below.

9.6 The capture_line Parameter

The capture_line parameter allows for an alternative or additional arrival criteria to be applied. When set to true, waypoint arrival will be achieved if the vehicle crosses the line perpendicular to the line segment approaching the waypoint from the previous waypoint. In the case of the first
waypoint, the "previous" waypoint is defined by the vehicle position when the behavior first becomes active.

When `capture_line` is set to `true`, both the capture line criteria and the capture and slip radius criteria apply. If either is satisfied, then arrival is achieved. If the `capture_line` criteria is set instead to `absolute`, then the capture and slip radius criteria is disabled, and only the capture line criteria is achieved.

When `capture_line` is set to `false`, only the capture and slip radius criteria are applied. A note of caution - setting the `capture_line` parameter to `absolute` essentially sets the capture and slip radius to zero. If the `capture_line` parameter is subsequently set to anything else (`true` or `false`), the capture and slip radius values also need to be explicitly re-set to non-zero values.

9.7 Track-line Following using the lead, lead_damper, and lead_to_start Parameters

By default the waypoint behavior will output a preference for the heading that is directly toward the next waypoint. By setting the `lead` parameter, the behavior will instead output a preference for the heading that keeps the vehicle closer to the track-line, or the line between the previous waypoint and the waypoint currently being driven to.

![Figure 25: The track-line mode](image)

The track-line mode: When in track line mode, the vehicle steers toward a point on the track line rather than simply toward the next waypoint. The steering-point is determined by the `lead` parameter. This is the distance from the perpendicular intersection point toward the next waypoint.

The distance specified by the `lead` parameter is based on the perpendicular intersection point on the track-line. This is the point that would make a perpendicular line to the track-line if the other point determining the perpendicular line were the current position of the vehicle. The distance specified by the `lead` parameter is the distance from the perpendicular intersection point toward the next waypoint, and defines an imaginary point on the track-line. The behavior outputs a heading preference based on this imaginary steering point. If the lead distance is greater than the distance to the next waypoint along the track-line, the imaginary steering point is simply the next waypoint.

Normally, when trackline following is enabled, it is enabled only between the first waypoint and all successor waypoints. When the parameter `lead_to_start` is set to true, trackline following is attempted even to the first waypoint, by defining a trackline from the vehicle’s present position.
when the behavior first enters the running mode.

If the lead parameter is enabled, it may be optionally used in conjunction with the lead_damper parameter. This parameter expresses a distance from the trackline in meters. When the vehicle is within this distance, the value of the lead parameter is stretched out toward the next waypoint to soften, or dampen, the approach to the trackline and reduce overshooting the trackline.

9.8 The wptflag Parameter

The wptflag parameter allows the user to specify flags to be posted upon the achievement of each waypoint, where waypoint achievement is defined by the normal means, depending on how the user has configured the capture_radius, slip_radius or capture_line parameters. The wptflags are in the same format as idle, end, active, or inactive flags defined generally for IvP behaviors, i.e., they consist of a MOOS variable and value. For example:

\[ wptflag = \text{STATION\_KEEP}=\text{true} \]

The flags also support a handful of macro expansions that allow the certain information about ownship or the next waypoint to be embedded in the posting that may not be available until run-time. These macros are:

- \$[X] or $(X)\]: Expanded to ownship’s x position in local coordinates.
- \$[Y] or $(Y)\]: Expanded to ownship’s y position in local coordinates.
- \$[NX] or $(NX)\]: Expanded to the x position of the next waypoint in local coordinates.
- \$[NY] or $(NY)\]: Expanded to the y position of the next waypoint in local coordinates.

Here are a couple examples of how the wptflag parameter may be useful in practice.

In the first case, in the example provided above, the mission may be configured to station-keep after arriving at each waypoint. This may be useful for a surface vehicle with a primary mission of collecting sensor measurements at periodic points in a survey area corresponding to waypoints. In this case, another station-keeping behavior is activated with the wptflag posting, which presumably temporarily idles the waypoint behavior while measurements are collected.

A second case utilizes the wptflag parameter to post the x-y position of the next waypoint in the list.

9.9 Variables Published by the Waypoint Behavior

The waypoint behavior publishes five variables for monitoring the performance of the behavior as it progresses: WPT_STAT, WPT_INDEX, CYCLE_INDEX, VIEW_POINT, VIEW_SEGLIST. The WPT_STAT contains information identifying the vehicle, the index of the current waypoint, the type of hits recorded for each waypoint, the distance to the current waypoint, and the estimated time of arrival to the current waypoint. Example output:

\[ \text{WPT\_STAT} = \text{"vname=alpha,behavior=traverse1,index=0,hits=10/11,dist=43,eta=23"} \]
The "hits=10/11" component in the above example indicates that, of the 11 waypoint arrivals achieved so far, 10 of them were achieved by meeting the capture radius criteria, and one of them was achieved by meeting the nonmonotonic radius criteria.

The WPT_INDEX variable simply publishes the index of the current waypoint. This is a bit redundant since this information is also in the WPT_STAT posting, but this variable is logged as a numerical variable, not a string, and facilitates the plotting of the index value as a step function in post mission analysis tools. The CYCLE_INDEX variable publishes the number of times the behavior has traversed the entire set of waypoints. The behavior may be configured to post the information in these three variables using alternative variables of the user’s liking:

post_mapping = WPT_STAT, MY_WPT_STATUS_VAR
post_mapping = WPT_INDEX, MY_WPT_INDEX_VAR
post_mapping = CYCLE_INDEX, MY_CYCLE_INDEX

or, to suppress the reports completely:

post_mapping = WPT_STAT, SILENT
post_mapping = WPT_INDEX, SILENT
post_mapping = CYCLE_INDEX, SILENT

Further posts to the MOOSDB can be configured to be made at the end of each cycle, that is, after reaching the last waypoint. Normally, if the repeat parameter remains at its default value of zero, then the end of a cycle and completing are identical and endflags can be used to post the desired information. However, when the behavior is configured to repeat the set of waypoints one or more times before completed, the cycleflag parameter may be used to post one or more variable-value pairs at the end of each cycle. Likewise, if the repeat parameter is zero, but the behavior is set with perpetual=true, the cycle flags will posted each new time that the behavior completes.

The VIEW POINT and VIEW_SEGLIST variables provide information consumable by a GUI application such as pMarineViewer or alogview for rendering the set of waypoints traversed by the behavior (VIEW_SEGLIST) and the behavior’s next waypoint (VIEW POINT). These two variables are responsible for the visual output in the Alpha Example Mission in Section 4 in Figure 7.

9.10 The Objective Function Produced by the Waypoint Behavior

The waypoint behavior produces a new objective function, at each iteration, over the variables speed and course/heading. The behavior can be configured to generate this objective function in one of two forms, either by coupling two independent one-variable functions, or by generating a single coupled function directly. The functions rendered in Figure 26 are built in the first manner.
9.11 Further Clarification on the repeat vs. perpetual Parameter

It’s worth clarifying the difference in usage and effect between the repeat parameter, which is specific to the Waypoint behavior, and the perpetual parameter which is defined for all helm behaviors. Normally when a behavior completes, it is entered into the completed stated, never again to be called upon by the helm. See Section 6.5.3 for more on behavior run states. It is up to the behavior implementor to decide what it means to be complete, and the implementor typically invokes the setComplete() function from within the code. See Section 7.5.1 for more on this function. In the case of the Waypoint behavior, completion by default occurs when the vehicle has hit all its waypoints. By setting perpetual=true the behavior, upon hitting all its waypoints, still invokes the setComplete() function, which causes it to post its endflags, but it does not enter the completed state. This feature is used, for example, in the Alpha example mission to allow the behavior to repeatedly return to the start point and be re-deployed, and later return to its start point again using the same Waypoint behavior.

The repeat parameter is used to change the criteria for completion. Whereas the normal criteria for completion is hitting all waypoints once, using repeat=N changes the criteria to be hitting all waypoints N+1 times. A few rules of thumb may be helpful in keeping things straight:

- When perpetual=false, the default setting, the behavior will permanently enter the completed state once it has hit all its waypoints.

- When perpetual=true and the behavior does not post endflags leading to its run conditions being unsatisfied, the behavior will repeat its waypoints indefinitely.

- When perpetual=true and it does indeed post endflags that lead to its run conditions being unsatisfied, it will remain in a running state until all its waypoints are hit. If the repeat parameter is used, it won’t post its endflags until it has repeated all waypoints the specified
number of times. During the course of traversal, the cycleflag posts will be made each time it has completed the set of waypoints. Upon completion, it will post its endflags and reset its cycle counter.

- By setting repeat=N, where N>0, the perpetual parameter is automatically set to true.

9.12 Visual Hints Defined for the Waypoint Behavior

Although the primary output of the Waypoint behavior is an IvP Function, a number of visual properties are also published for convenience in mission monitoring. This includes (a) the set of waypoints, (b) the point the behavior is presently progressing toward, and (c) the trackpoint, if trackpoint following is enabled. These visual artifacts have default properties in size and color that may be altered to the user’s preferences. These preferences are configurable through the visual_hints parameter. Each parameter below is used in the following way by example:

```
visual_hints = vertex_size=3, edge_size=2
visual_hints = vertex_color=khaki
```

- **vertex_size**: The size of vertices rendered in the loiter polygon. The default is 1.
- **edge_size**: The width of edges rendered in the loiter polygon. The default is 1.
- **vertex_color**: The color of vertices rendered in the loiter polygon. The default is "dodger_blue".
- **edge_color**: The color of edges rendered in the loiter polygon. The default is "white".
- **label_color**: The color of label for set of waypoints. The default is "white".
- **nextpt_color**: The color of the point rendered as the present next waypoint. The default is "yellow".
- **nextpt_lcolor**: The color of the label for the point rendered as the present next waypoint. The default is "aqua".
- **nextpt_vertex_size**: The size of the vertex for the point rendered as the present next waypoint. The default is 1.

Rendering of vertices or the next waypoint may be shut off with a size of zero, and labels may be shut off with the special color "invisible". For a list of legal colors, see Appendix C.
10 The OpRegion Behavior

This behavior provides four different types of safety functionality, (a) a boundary box given by a convex polygon in the x-y or lat-lon plane, (b) an overall timeout, (c) a depth limit, (d) an altitude limit. The behavior does not produce an objective function to influence the vehicle to avoid violating these safety constraints. This behavior merely monitors the constraints and posts an error which results in the posting of all-stop commands, which could put the vehicle into the `PARK` state if the helm is configured with `park_on_allstop` is set to true. More on this parameter can be found in Section 5.3.

10.1 Configuration Parameters

Listing 10.1: Configuration Parameters Common to All Behaviors.

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the `active` state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the `idle` state. Section 7.2.6.
- **duration_reset**: A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the `idle` state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is not in the `active` state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
- **nostarve**: Allows a behavior to assert a maximum staleness for a MOOS variable. Section 7.2.9.
- **perpetual**: If true allows the behavior to to run even after it has completed. Section 7.2.7.
- **post_mapping**: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- **priority**: The priority weight of the behavior. Section 7.2.3.
- **put**: Same as `priority`.
- **runflag**: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 10.2: Configuration Parameters for the OpRegion Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>breached_altitude_flag</td>
<td>A MOOS variable-value pair to be posted when or if the vehicle breaches the altitude limit set by this behavior. New with Release 14.7.1. Section 10.6.</td>
</tr>
<tr>
<td>breached_depth_flag</td>
<td>A MOOS variable-value pair to be posted when or if the vehicle breaches the depth limit set by this behavior. New with Release 14.7.1. Section 10.6.</td>
</tr>
<tr>
<td>breached_poly_flag</td>
<td>A MOOS variable-value pair to be posted when or if the vehicle breaches the convex polygon op area limit set by this behavior. New with Release 14.7.1. Section 10.6.</td>
</tr>
<tr>
<td>breached_time_flag</td>
<td>A MOOS variable-value pair to be posted when or if the vehicle breaches the time limit set by this behavior. New with Release 14.7.1. Section 10.6.</td>
</tr>
<tr>
<td>max_time</td>
<td>The max allowable time in seconds. Section 10.4.</td>
</tr>
<tr>
<td>max_depth</td>
<td>The max allowable depth in meters. Section 10.4.</td>
</tr>
<tr>
<td>min_altitude</td>
<td>The min allowable altitude in meters. Section 10.4.</td>
</tr>
<tr>
<td>opregion_poly_var</td>
<td>The name of a MOOS variable to which the polygon points will be posted. By default is empty, meaning no such post is made.</td>
</tr>
<tr>
<td>polygon</td>
<td>The lat-lon area the vehicle is restricted to stay within. Section 10.3.</td>
</tr>
<tr>
<td>reset_var</td>
<td>A MOOS variable to allow behavior reset when or if the behavior is breached. New with Release 14.7.1. Section 10.7.</td>
</tr>
<tr>
<td>time_remaining_var</td>
<td>A MOOS variable to publish the remaining time left if max_time is specified. If this variable is specified, but the maximum time is not enforced, then it will publish with a value of -1. New with Release 14.7.1.</td>
</tr>
<tr>
<td>trigger_entry_time</td>
<td>The time required for the vehicle to have been within the polygon region before triggering the polygon requirement. Section 10.3.</td>
</tr>
<tr>
<td>trigger_exit_time</td>
<td>The time required to have been outside the polygon before declaring a polygon containment failure. Section 10.3.</td>
</tr>
<tr>
<td>visual_hints</td>
<td>Hints for visual properties in variables posted intended for rendering.</td>
</tr>
</tbody>
</table>

Listing 10.3: Example Configuration Block.
Behavior = BHV_OpRegion
{
   // General Behavior Parameters
   // ---------------------------
   name = op_region // example
   pwt = 100 // default
   condition = SAFETY=true // example
   updates = OPREGION_UPDATES // example

   // Parameters specific to this behavior
   // ------------------------------------
   max_time = 0 // default (seconds)
   max_depth = 0 // default (meters)
   min_altitude = 0 // default (meters)
   reset_var = OPREGION_RESET // example
   trigger_entry_time = 1 // default (seconds)
   trigger_exit_time = 0.5 // default (seconds)

   polygon = pts={-80,-50:-30,-175:150,-100:95,25}, label=area_x

   breached_altitude_flag = SAY_MOOS = Sir, the min altitude has been exceeded
   breached_depth_flag = SAY_MOOS = Sir, the max depth has been exceeded
   breached_poly_flag = SAY_MOOS = Sir, the op region has been violated
   breached_time_flag = SAY_MOOS = Sir, the maximum mission time has been exceeded

   visual_hints = vertex_color = brown // default
   visual_hints = vertex_size = 3 // default
   visual_hints = edge_color = aqua // default
   visual_hints = edge_size = 1 // default
}

10.2 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be supressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.8.

- **OPREG_TRAJECTORY_PERIM_DIST**: Distance, in meters, to the perimeter on the current trajectory.
- **OPREG_TRAJECTORY_PERIMETA**: Time to the perimeter on the current speed and trajectory.
- **OPREG_ABSOLUTE_PERIM_DIST**: Distance, in meters, to the op-region in any direction.
- **OPREG_ABSOLUTE_PERIMETA**: Time to the op-region perimeter at current speed in any direction.
- **OPREG_TIME_REMAINING**: Time, in seconds, until the max_time upper bound is exceeded.
- **VIEW_POLYGON**: A visual cue for rendering the polygon containment region.

10.3 Safety Checking Applied to an Operation Region

One safety check performed by the OpRegion behavior is to ensure that the vehicle remains in an operation region defined by a convex polygon in the x-y plane.

- **polygon**: A colon separated list of x,y pairs given as points in space, typically meters. A pair
given by "label,string" can associate an optional label with the point list. The collection of points must be a convex polygon. A check for convexity is done upon helm/behavior start-up. Behavior initialization will fail if it is not convex. If no polygon is provided, no X,Y checks are made.

- **trigger_entry_time:** The amount of time required for the vehicle to have been within the polygon containment region before triggering the polygon containment requirement. This is useful when launching vehicles from a dock structure such as the MIT Sailing Pavilion. The default setting is zero meaning the polygon containment requirement is active immediately.

- **trigger_exit_time:** The amount of time required to have been outside the polygon containment region before declaring a polygon containment failure. This is useful if the vehicle NAV.X and NAV.Y position is based on a sensor without outlier detection. The kayaks, for example, are often relying solely on GPS which occasionally emits an outlier well out of the containment region. By setting this value high enough, outliers are ignored. Each time a recorded position is contained within the polygon region, the clock is set to zero. The default setting is zero, meaning the very first detection outside the polygon will result in a polygon containment error.

10.4 Safety Limits on Operation Time, Depth, and Vehicle Altitude

- **max_time:** The maximum allowable time (in seconds) that the helm is allowed to run. The clock starts when the pHelmIvP process first takes control, i.e., enters the DRIVE state. If no maximum time is specified, then no time checks are made.

- **max_depth:** The maximum allowable depth of the vehicle (in meters). If no depth is provided, no depth checks are made.

- **min_altitude:** The minimum allowable altitude of the vehicle (in meters). If no altitude is provided, no altitude checks are made.

10.5 Variables Published by the OpRegion Behavior

The behavior also produces a set of status variables regarding the vehicle position with respect to the containment region. Since a violation of this constraint results in a vehicle full-stop and the helm relinquishing control, other behaviors or MOOS processes may want to take measures to avoid it. These status variables provide information on the position and estimated time between the vehicle and the perimeter, based both on the absolute position as well as the current vehicle trajectory. See Figure 27.
The four variables produced by the behavior (and posted to the MOOSDB by the Helm) are:

- **OPREG_TRAJECTORY_PERIM_DIST**: The distance (in meters) between the current vehicle position to the perimeter of the polygon containment region (given by the `polygon` parameter), based on the vehicle remaining on the current trajectory.

- **OPREG_TRAJECTORY_PERIM_eta**: The amount of time (in seconds) needed for the vehicle to reach the perimeter of the polygon containment region (given by the `polygon` parameter), based on the vehicle remaining on the current trajectory.

- **OPREG_ABSOLUTE_PERIM_DIST**: The distance (in meters) between the current vehicle position to the perimeter of the polygon containment region (given by the `polygon` parameter), regardless of the current vehicle trajectory.

- **OPREG_ABSOLUTE_PERIM_eta**: The amount of time (in seconds) needed for the vehicle to reach the perimeter of the polygon containment region (given by the `polygon` parameter), regardless of the current vehicle trajectory. Calculated on the maximum vehicle speed.

### 10.6 Publishing Breach Flags

The behavior may be configured with four different types of *breach flags*:

- The `breached_altitude_flag` names a MOOS variable-value pair to be posted when or if the minimum altitude (if set) is exceeded.
- The `breached_depth_flag` names a MOOS variable-value pair to be posted when or if the maximum depth (if set) is exceeded.
- The `breached_poly_flag` names a MOOS variable-value pair to be posted when or if the vehicle exits the polygon containment area (if set).
• The `breached_time_flag` names a MOOS variable-value pair to be posted when or if the maximum mission time (if set) is exceeded.

Each type of flag may be configured with more than one posting using separate lines in the configuration block. The flags are posted once upon the initial breach, but may be posted again later if the behavior is re-set and the breach re-occurs.

10.7 Resetting a Breach Condition

The behavior may be configured to accept a reset after one of the breach conditions occur. This is done by naming a MOOS variable using the `reset_var` parameter. Upon receipt of this mail, the behavior returns to the state as when it was first spawned at launch time.

For example, if the behavior were configured with:

```
reset_var = OPR_RESET
```

then the following postings would reset the behavior if the vehicle breached the OpRegion polygon:

```
OPR_RESET = poly
```

Similarly, if the OpRegion behavior were breached due to a depth violation, posting `OPR_RESET=depth` would reset the behavior. Posting `OPR_RESET=time` would reset the behavior after breaching its max time limit. Posting `OPR_RESET=altitude` would reset the behavior after breaching its min altitude limit.
11 The Loiter Behavior

A behavior for transiting to and repeatedly traversing a set of waypoints forming a convex polygon. A similar effect can be achieved with the Waypoint behavior but this behavior assumes a set of waypoints forming a convex polygon to exploit certain useful algorithms discussed below. It also utilizes the non-monotonic arrival criteria used in the Waypoint behavior to avoid loop-backs upon waypoint near-misses. It also robustly handles dynamic exit and re-entry modes when or if the vehicle diverges from the loiter region due to external events. It is dynamically reconfigurable to allow a mission control module to repeatedly reassign the vehicle to different loiter regions by using a single persistent instance of the behavior.

11.1 Configuration Parameters

Listing 11.1: Configuration Parameters Common to All Behaviors.

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the idle state. Section 7.2.6.
- **duration_reset**: A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
- **nostarve**: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.
- **perpetual**: If true allows the behavior to run even after it has completed. Section 7.2.7.
- **post_mapping**: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- **priority**: The priority weight of the behavior. Section 7.2.3.
- **pwt**: Same as priority.
- **runflag**: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 11.2: Configuration Parameters for the Loiter Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire_dist</td>
<td>Distance to polygon outside which the behavior will be in &quot;acquire&quot; mode.</td>
</tr>
<tr>
<td>capture_radius</td>
<td>The radius tolerance, in meters, for satisfying the arrival at a point.</td>
</tr>
<tr>
<td>center_activate</td>
<td>If true, center of polygon is set to present position when behavior activates.</td>
</tr>
<tr>
<td>center_assign</td>
<td>An x,y pair, in meters, indicating a (new) center of the loiter polygon.</td>
</tr>
<tr>
<td>clockwise</td>
<td>If true, loitering is done clockwise (the default setting).</td>
</tr>
<tr>
<td>polygon</td>
<td>Polygon about which the vehicle will traverse and loiter.</td>
</tr>
<tr>
<td>post_suffix</td>
<td>A string to add as a suffix to variables posted by this behaviors.</td>
</tr>
<tr>
<td>radius</td>
<td>An alias for capture_radius.</td>
</tr>
<tr>
<td>slip_radius</td>
<td>An &quot;outer&quot; capture radius. Arrival declared when the vehicle is in this range</td>
</tr>
<tr>
<td>speed</td>
<td>Speed in meters per second.</td>
</tr>
<tr>
<td>speed_alt</td>
<td>An alternative desired speed (m/s) at which the vehicle travels through the</td>
</tr>
<tr>
<td></td>
<td>points. Applies only when the use_alt_speed parameter is set to true.</td>
</tr>
<tr>
<td>spiral_factor</td>
<td>The degree of spiraling when activated at the center of the polygon.</td>
</tr>
<tr>
<td>use_alt_speed</td>
<td>If true then attempt to use the alternate speed set with the speed_alt</td>
</tr>
<tr>
<td></td>
<td>parameter, if that speed is set to a non-negative value. The default is -1,</td>
</tr>
<tr>
<td></td>
<td>which indicates internally that it has not been set.</td>
</tr>
<tr>
<td>visual_hints</td>
<td>Hints for visual properties in variables posted intended for rendering.</td>
</tr>
<tr>
<td>xcenter_assign</td>
<td>A x-value, in meters, indicating a (new) x-position of the loiter polygon.</td>
</tr>
<tr>
<td>ycenter_assign</td>
<td>A y-value, in meters, indicating a (new) y-position of the loiter polygon.</td>
</tr>
</tbody>
</table>

(Introduced after release 15.5.)

Listing 11.3: Example Configuration Block.
Behavior = BHV_Loiter
{
    // General Behavior Parameters
    // ---------------------------
    name = transit       // example
    pwt = 100            // default
    condition = MODE==LOITERING // example
    updates = LOITER_UPDATES // example

    // Parameters specific to this behavior
    // ------------------------------------
    acquire_dist = 10    // default
    capture_radius = 3   // default
    center_activate = false // default
    clockwise = true     // default
    slip_radius = 15     // default
    speed = 0            // default
    spiral_factor = -2   // default

    polygon = radial:: x=5,y=8,radius=20,pts=8 // example
    post_suffix = HENRY  // example

    center_assign = 40,50 // example
    xcenter_assign = 40  // example
    ycenter_assign = 50  // example

    visual_hints = vertex_size = 1    // default
    visual_hints = edge_size = 1      // default
    visual_hints = vertex_color = dodger_blue // default
    visual_hints = edge_color = white  // default
    visual_hints = nextpt_color = yellow // default
    visual_hints = nextpt_lcolor = aqua // default
    visual_hints = nextpt_vertex_size = 5 // default
    visual_hints = label = zone3      // example
}

11.2 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be suppressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.8.

- **LOITER_ACQUIRE**: Posts 1 when in the "acquire" mode, 0 otherwise.
- **LOITER_DIST_TO_POLY**: Current distance, in meters, to the loiter polygon.
- **LOITER_ETA_TO_POLY**: Estimated time of arrival to the polygon at present speed and trajectory.
- **LOITER_INDEX**: The index of the vertex in the loiter polygon currently heading toward.
- **LOITER_MODE**: A string indicating details of the acquire mode, e.g., "acquiring_external".
- **LOITER_REPORT**: A status string with current mode, current vertex, and prior arrivals.
- **VIEW_POINT**: A visual cue for rendering the next point in the loiter polygon.
- **VIEW_POLYGON**: A visual cue for rendering the loiter polygon.
The following are some examples:

```plaintext
LOITER_REPORT = index=5,capture_hits=51,nonmono_hits=0,acquire_mode=false
LOITER_INDEX = 5
LOITER_ACQUIRE = 1
LOITER_DIST_TO_POLY = 2.2
LOITER_ETA_TO_POLY = 294.4
LOITER_MODE = stable
VIEW_POLYGON = edge_size,0.0:vertex_size,0.0:0,-50:0,-150:150,-150:150,-50
```

11.3 Detailed Discussions on Loiter Behavior Parameters

11.3.1 The **polygon** Parameter for Setting the Loiter Region

The Loiter behavior is configured with a *loiter region*, defined by a convex polygon. The behavior will influence the vehicle to repeatedly traverse the set of points on the polygon. The polygon is specified typically in the behavior configuration block. Any string that properly defines a convex polygon is acceptable. The following two configuration lines, for example, will result in the same polygon.

```plaintext
polygon = 20,-40:40,-75:20,-110:-20,-110:-40,-75:-20,-40:label,Lima
polygon = format=radial, x=0, y=-75, radius=40, pts=6, snap=1, label=Lima
```

Each would produce the polygon shown in Figure 28:

![Figure 28: A typical loiter polygon with six vertices.](image)

The shape and position polygon may be altered dynamically (after the helm is launched and running) in one of two manners by either specifying new parameters explicitly, or by tying the loiter position to the vehicle position when the loiter behavior enters the running behavior mode.

11.3.2 The **updates** Parameter for Updating the Loiter Region

In the first case, altering the polygon parameter dynamically is accomplished by using the standard **updates** parameter described in Section 7.2.5. For example, the behavior may be configured to

```plaintext
123
```
receive new updates by adding the following line to its configuration block:

```
updates = NEW_CONFIG
```

Its position may be then moved 75 meters North by posting the following to the MOOSDB:

```
NEW_CONFIG = "polygon = format=radial, x=0, y=0, radius=40, pts=6, snap=1, label=Lima"
```

Alternatively, if one wants to simply move the polygon to a new \((x,y)\) position, the Loiter behavior also implements the `center_assign` configuration parameter. The same shift as above can be made without the source making the post having to know anything about the other parameters of the polygon:

```
NEW_CONFIG = "center_assign = 0,0"
```

Likewise, if one simply wants to move the polygon in either the \(x\) or \(y\) direction without knowing anything about the position in the other direction, the Loiter behavior implements the `xcenter_assign` and `ycenter_assign` parameters. Thus the above shift could also be accomplished without the source making the post knowing anything about the current \(x\) position of the polygon:

```
NEW_CONFIG = "ycenter_assign = 0"
```

### 11.3.3 The `center_activate` Parameter for Using the Current Vehicle Position

The Loiter behavior may also be configured to reset the \((x,y)\) center position of the loiter polygon to the present position of the vehicle at the very moment the behavior transitions from the idle to the running state. See Section 6.5.3 for more on behavior run states. This configuration is declared with the following line in the behavior configuration block:

```
center_activate = true
```

This setting is useful if one wants to, for example, send a command to the vehicle to exit some other mission mode and simply loiter at its present location until further notice. Of course this configuration as well, may also be dynamically toggled through a variable specified in the `updates` parameter.

### 11.3.4 The `speed` Parameter for Setting the Loiter Speed

The desired speed, in meters/second, at which the vehicle travels through the points.

### 11.3.5 The `speed_alt` and `use_alt_speed` Parameters

When the `use_alt_speed` parameter is set to "true", the loiter behavior will use speed set in the `speed_alt` parameter, if it has been set. By default `use_alt_speed` is "false" and `speed_alt` is set to \(-1\). This provides a convenient way for other behaviors or apps to periodically influence the loiter behavior to a different speed. The other behavior or app does not need to read, remember and reset the loiter behavior back to its original speed, but can instead just toggle `use_alt_speed` to `false` to return the loiter back to its original speed.
11.3.6 The spiral_factor Parameter

A value in the range \([0, 100]\) indicating the degree to which the vehicle will spiral out to the polygon if started on the inside. A setting of zero indicates it will move directly to the first polygon vertex. This parameter only relates to the situation of starting inside the polygon. The default value is 98.

11.3.7 The acquire_dist Parameter

The acquire_dist parameter is the distance in meters between the vehicle and the polygon that will trigger the vehicle to return to acquire mode. This notion applies to the case where the vehicle is both inside and outside the polygon. The re-acquire algorithms are different however. The default is 10 meters.

11.3.8 The xcenter_assign and ycenter_assign Parameters

11.3.9 The capture_radius and slip_radius Parameters

radius: The radius tolerance, in meters, for satisfying the arrival at a waypoint. As soon as the vehicle is within this distance to the waypoint the waypoint behavior begins operating on the next waypoint in the sequence, or completes and posts its endflags if there are no more waypoints.

slip_radius: As the vehicle progresses toward a waypoint, the sequence of measured distances to the waypoint decreases monotonically. The sequence becomes non-monotonic when it hits its waypoint or when there is a near-miss of the waypoint arrival radius. The slip_radius, a.k.a, the nm_radius, short for non-monotonic radius is an arrival radius distance within which a detection of increasing distances to the waypoint is interpreted as a waypoint arrival. This distance would have to be larger than the arrival radius to have any effect (see Figure 24). As a rule of thumb, a distance of twice the arrival radius is practical.

11.3.10 The post_suffix Parameter

The post_suffix parameter names a string to be added as a suffix to each of the status variables posted by the behavior (LOITER_REPORT, LOITER_INDEX, LOITER_ACQUIRE, LOITER_DIST2POLY). By default, the suffix is the empty string and the variables will be posted as above. When multiple Loiter behaviors are configured in the helm it may help to distinguish the posted variables by a suffix. A given suffix of "FOO" would result in the posting of LOITER_INDEX_FOO for example. The extra ’_’ character is inserted automatically.

11.3.11 The clockwise Parameter for Setting the Loiter Direction

The user may configure the direction the vehicle traverses the loiter polygon by setting the parameter:

\[
\text{clockwise} = \langle\text{value}\rangle
\]

The possible case insensitive settings for \(<\text{value}\rangle\) are "true", "false", "best". In the first two cases, the directions are explicitly set and will not vary regardless of the pose of the vehicle with respect to the polygon. In the case where clockwise=best, the direction is re-evaluated once, whenever the behavior run state transitions from idle to running. Thus the direction depends on the pose relative
position to the polygon at that particular point in time. This is shown in the lower case in Figure 29 below.

![Figure 29: The Loiter Direction. The polygon traversal direction is determined by the clockwise configuration parameter. It may be explicitly set as shown on the top, or determined at run time when the behavior becomes non-idle as shown on the bottom.](image)

Regardless of the prevailing direction, as the vehicle is transiting to the loiter polygon, the behavior will influence the vehicle toward the vertex that allows for the smoothest entry, given the chosen direction. If the polygon were a perfect circle, the vehicle would approach on one of the two tangent lines.

11.3.12 The visual_hints Parameter

Although the primary output of the Loiter behavior an IvP Function, a number of visual properties are also published for convenience in mission monitoring. This includes both the loiter polygon and the point on the polygon the behavior is presently progressing toward. These visual artifacts have default properties in size and color that may be altered to the user’s preferences. These preferences are configurable through the visual_hints parameter. Each parameter below is used in the following way by example:

```
visual_hints = vertex_size=3, edge_size=2
visual_hints = vertex_color=khaki
```

- **vertex_size**: The size of vertices rendered in the loiter polygon. The default is 1.
- **edge_size**: The width of edges rendered in the loiter polygon. The default is 1.
- **vertex_color**: The color of vertices rendered in the loiter polygon. The default is ”dodger_blue”.

126
• **edge_color**: The color of edges rendered in the loiter polygon. The default is "white".
• **nextpt_color**: The color of the point rendered as the present next waypoint. The default is "yellow".
• **nextpt_lcolor**: The color of the label for the point rendered as the present next waypoint. The default is "aqua".
• **label**: The label of rendered for the loiter polygon.

Rendering of vertices or the next waypoint may be shut off with a size of zero, and labels may be shut off with the special color "invisible". For a list of legal colors, see Appendix C.

### 11.4 The Loiter Acquisition Mode

When the Loiter behavior is running (non-idle), it behaves differently depending on where the vehicle is in relation to the loiter polygon. The behavior has a notion of (a) when it is progressing in a "stable" manner around the polygon and (b) when it is trying to move the vehicle back to (a). The difference between the two is solely determined by whether or not the vehicle is within some acquire distance from the loiter polygon. This distance is set with the behavior configuration parameter:

\[
\text{acquire\_dist} = \text{<distance>}
\]

The `<distance>` component must simply be a non-negative value. A value of zero should work fine, but essentially disables some useful ways in which the behavior may be coordinated with other parts of the mission. This relationship is shown in Figure 30.

![Figure 30: The Loiter Polygon Zones](image)

Figure 30: **The Loiter Polygon Zones**: The Loiter behavior regards itself to be in one of three distinct possible zones relative to the polygon boundary - stable, internal, external, depending on the setting of the acquire_dist parameter as shown.

When the vehicle is not within the stable zone, it is trying to acquire the polygon in one of four distinct manners, depending on whether (a) the vehicle is internal or external and (b) whether it is "acquiring" the polygon for the first time, or whether it is "recovering" from having drifted out of the stable zone. The idea is depicted in Figure 31.
The current loiter mode is published by the behavior in the MOOS variable `LOITER_MODE`. As with any MOOS variable published by a behavior, this may be remapped to a different variable of the user's liking with the `post_mapping` configuration parameter.

When the behavior is in any loiter mode other than the stable mode, it recalculates on each iteration which vertex it should be heading toward, the approach vertex. In the case of an external approach, the chosen vertex should remain steady unless there are external forces such as wind or current, or if the vehicle changes its aspect to the polygon significantly as it is executing a turn. In the case of an internal approach, the approach vertex will likely change during the approach, outward toward the polygon boundary, creating a pseudo outward spiral trajectory. Note that re-evaluating the approach vertex is not the same as re-evaluating the traversal direction of the polygon. The latter is only re-evaluated dynamically if the behavior is configured with `clockwise=best`, and then only when the behavior becomes non-idle.

The circumstance most common for triggering the acquire mode is the initial assignment to the vehicle to loiter at a new given region in the X,Y plane. This assignment could occur while the vehicle happens to already be within the polygon for a number of reasons. Furthermore, the vehicle could be driven off the polygon loiter trajectory due to environmental (wind or current) forces or the temporary dominance of other vehicle behaviors such as collision avoidance or tracking of another vehicle.

Once the behavior enters the acquire mode, it remains in this mode until arriving at the first waypoint (defined by the arrival and non-monotonic radii settings), after which it switches to normal mode until the acquire mode is re-triggered or the behavior run conditions are no longer met. There is currently no "complete" condition for this behavior other than a time-out which is defined for all behaviors.
12 The PeriodicSpeed Behavior

This behavior will periodically influence the speed of the vehicle while remaining neutral at other times. The timing is specified by a given period in which the influence is on (busy), and a period specifying when the influence if off (lazy), as depicted in Figure 32.

![Figure 32: Busy and Lazy Modes](image)

Figure 32: **Busy and Lazy Modes**: In the busy mode the behavior will produce an objective function defined over speed that will potentially influence the speed of the vehicle. In the lazy mode, it simply will not produce an objective function.

It was conceived for use on an AUV equipped with an acoustic modem to periodically slow the vehicle to reduce self-noise and reduce communication difficulty. One can also specify a flag (a MOOS variable and value) to be posted at the start of the period to prompt an outside action such as the start of communication attempts.

12.1 Configuration Parameters

*Listing 12.1: Configuration Parameters Common to All Behaviors.*

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the *active* state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the *idle* state. Section 7.2.6.
- **duration_reset**: A variable-pair such as **MY_RESET=true**, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the *idle* state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is *not* in the *active* state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
nostarve: Allows a behavior to assert a maximum staleness for a MOOS variable. Section 7.2.9.

perpetual: If true allows the behavior to run even after it has completed. Section 7.2.7.

post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.

priority: The priority weight of the behavior. Section 7.2.3.

pwt: Same as priority.

runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.

spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

**Listing 12.2: Configuration Parameters for the Periodic Speed Behavior.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basewidth:</td>
<td>The width of the ZAIC basewidth, in m/sec, in the IvP function.</td>
</tr>
<tr>
<td>initially_busy:</td>
<td>If true the initial state is busy. The default is false.</td>
</tr>
<tr>
<td>peakwidth:</td>
<td>The width of the ZAIC peakwidth, in m/sec in the IvP function.</td>
</tr>
<tr>
<td>period_busy:</td>
<td>The duration of the busy period, in seconds.</td>
</tr>
<tr>
<td>period_lazy:</td>
<td>The duration of the lazy period, in seconds.</td>
</tr>
<tr>
<td>period_speed:</td>
<td>The desired speed, in m/sec, in the IvP function.</td>
</tr>
<tr>
<td>reset_upon_running:</td>
<td>Initial conditions reset upon entering the running state. Default is true.</td>
</tr>
<tr>
<td>summit_delta:</td>
<td>The extent of the ZAIC summit delta in the IvP function.</td>
</tr>
</tbody>
</table>

**Listing 12.3: Example Configuration Block.**
Behavior = BHV_PeriodicSpeed
{
    // General Behavior Parameters
    // ---------------------------
    name = transit // example
    pwt = 100 // default
    condition = MODE==TRANSITING // example
    updates = PSPD_UPDATES // example

    // Parameters specific to this behavior
    // ------------------------------------
    basewidth = 0 // default
    initially_busy = false // default
    peakwidth = 0 // default
    period_busy = 0 // default
    period_lazy = 0 // default
    period_speed = 0 // default
    reset_upon_running = true // default
    summit_delta = 25 // default
}

12.2 Variables Published

Listing 12.4: Variables Published the Periodic Speed Behavior.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS_PENDING_BUSY</td>
<td>Publishes the amount of time in seconds until the behavior reaches the busy mode. During the busy mode this value is zero, and it resets to the value given by the period_lazy parameter upon transitioning into the lazy mode.</td>
</tr>
<tr>
<td>PS_PENDING_LAZY</td>
<td>Publishes the amount of time in seconds until the behavior reaches the lazy mode. During the lazy mode this value is zero, and it resets to the value given by period_busy upon transitioning back into the busy mode. To reduce posting volume, the values posted will be rounded to the nearest second until less than one second remains, after which fractions are posted.</td>
</tr>
<tr>
<td>PS_BUSY_COUNT</td>
<td>This variable is posted by the behavior each time it enters the busy mode. The value is an integer indicating the number of times it has entered the busy mode, posting zero initially.</td>
</tr>
</tbody>
</table>

12.3 State Transition Policy and Initial Condition Parameters

The behavior alternates between one of two modes, the busy mode or the lazy mode. In the former, it will produce an objective function over the speed decision variable, and in the latter mode it will simply refrain from producing the objective function. Note that these modes are different from the general behavior run states described in Section 6.5.3. If this behavior is idle, i.e., has not met the conditions for being in the running state, it will not generate an objective function regardless of whether it is in the busy or lazy mode.

When the behavior enters the busy mode, it resets a timer which counts down from period_busy seconds, after which it enters the lazy mode. When it enters the lazy mode, it resets a timer which counts down from period_lazy seconds, after which it goes back to the busy mode. By default the
behavior is initially in the lazy mode, but it can be configured in the opposite manner by setting \texttt{initially\_busy} to \texttt{true}.

By default, when \texttt{reset\_upon\_running} is true, each time the behavior enters the running state, the busy/lazy mode is set to its initial value, and all timers are reset to their initial values. This is true regardless of whether it is entering the running state upon initial helm engagement, or due to transitioning from the idle state. This can be changed by setting \texttt{reset\_upon\_running} to false. In this case the timers are counting down immediately upon helm engagement and continue to do so regardless of the behavior run state.
13 The AvoidObstacle Behavior

The AvoidObstacleV21 behavior will produce IvP objective functions designed to avoid obstacles (and near collisions) with given specified obstacles. As newer versions of this behavior emerge, they will take on a suffix that reflects the release date. The discussion here reflects the functionality of the AvoidObstacleV21 version, but we will refer to this behavior simply as the AvoidObstacle behavior.

13.1 Overview

The AvoidObstacle behavior acts upon a single obstacle given by a convex polygon, Figure 33. When a robot has multiple obstacles, multiple versions of the behavior are spawned, one for each obstacle. During the course of a mission, it is expected that multiple such behavior instances will be spawned and deleted.

The AvoidObstacle behavior is typically configured as a template for spawning instances as obstacles emerge. In less common cases, one or more static behaviors may be configured, instantiated at mission launch time, with an obstacle of a known position and location. Both use cases will be described. In the case of dynamically spawned behaviors, the helm and the AvoidObstacle behavior work in coordination with an obstacle manager which sits between the helm and sensor processing applications to reason about obstacles and post messages at the appropriate time to the helm, for generating new AvoidObstacle behaviors. The obstacle manager currently used is another MOOS app called pObstacleMgr and is documented separately. The interface that triggers AvoidObstacle instances will be described.

![Figure 33: The AvoidObstacle Behavior](image)

The avoidObstacle behavior is not a path planner. It is a behavior that works in conjunction with a planned path, typically being executed by the Waypoint behavior. The AvoidObstacle behavior is typically used for avoid locally sensed obstacles like buoys, stationary vessels, or any other hazard that may be detected or known about beforehand. The AvoidObstacle behavior, like any IvP Helm behavior, is capable of accepting dynamic updates. This includes obstacle size and location. So the
AvoidObstacle behavior is capable of dealing with moving obstacles including vessels. However, when a vessel is known to be a vessel, and has a known position, heading, and speed, the AvoidCollision or AvdColregs behaviors are more appropriate.

### 13.2 The Nature and Origin of Obstacles

The AvoidObstacle behavior may be configured *statically* or *dynamically*. In a static behavior the obstacle is provided in the mission configuration using the `polygon` configuration parameter. For example:

```
polygon = 60,-40 : 60,-160 : 150,-160 : 180,-100 : 150,-40
```

The vertices may also be specified indirectly using one more supported patterns. For example an octogon polygon, perhaps for avoiding a 1 meter sized buoy at a local coordinates (45,90) could be given as:

```
polygon = format=radial, x=45, y=90, radius=2, pts=8
```

In a dynamic behavior, the polygon specification is typically provided by an external source, typically the obstacle manager. In the mission file this behavior leaves the `polygon` parameter unspecified, but does specify a MOOS variable for receiving updates. For example:

```
templating = spawn
updates = OBSTACLE_ALERT
```

A new AvoidObstacle behavior

In both cases, they may be altered during run time.

### 13.3 Configuration Parameters

*Listing 13.1: Configuration Parameters Common to All Behaviors.*

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the *active* state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the *idle* state. Section 7.2.6.
- **duration_reset**: A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
idleflag: A MOOS variable-value pair posted when the behavior is in the *idle* state. Section 6.5.4.

inactiveflag: A MOOS variable-value posted when the behavior is *not* in the *active* state. Section 6.5.4.

ame: The (unique) name of the behavior. Section 7.2.2.

nostarve: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.

perpetual: If true allows the behavior to to run even after it has completed. Section 7.2.7.

post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.

priority: The priority weight of the behavior. Section 7.2.3.

pwt: Same as *priority*.

runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.

spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

### Listing 13.2: Configuration Parameters for the AvoidObstacle Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>completed_dist:</td>
<td>Range to contact outside of which the behavior completes and dies. The default is 500 meters.</td>
</tr>
<tr>
<td>max_util_cpa_dist:</td>
<td>Range to contact outside which a considered maneuver will have max utility. Section ??. The default is 75 meters</td>
</tr>
<tr>
<td>min_util_cpa_dist:</td>
<td>Range to contact within which a considered maneuver will have min utility. Section ??. The default is 10 meters.</td>
</tr>
<tr>
<td>pwt_inner_dist:</td>
<td>Range to contact within which the behavior has maximum priority weight. Section 13.6. The default is 50 meters.</td>
</tr>
<tr>
<td>pwt_outer_dist:</td>
<td>Range to contact outside which the behavior has zero priority weight. Section 13.6. The default is 200 meters.</td>
</tr>
</tbody>
</table>
| use_refinery: | If true, the behavior will produce an optimized objective function that is faster to produce, uses a smaller memory footprint, and contributes to faster helm solution time. The default is false, simply for continuity with prior releases, but there is no downside to enabling this feature. Section ??.
| polygon: | A convex polygon representing the obstacle. Section ??.
| poly: | Same as *polygon*.
| rng_flag: | A flag to be posted on all iterations. It may be conditioned on a range threshold. Section ??.
| cpa_flag: | A flag to be posted upon reaching the closest point of approach to the obstacle. Section ??.

135
**visual_hints:** A request to override the default visual parameters of the rendered obstacle or its buffer region. Section ??.

**allowable_ttc:** The allowable time-to-collision, in seconds, within which a candidate trajectory will begin to be penalized. Section ??.

### Listing 13.3: Example Configuration Block.

```plaintext
Behavior = BHV_AvoidObstacleV21
{
// General Behavior Parameters
// ---------------------------
name = avdob_
pwt = 300
condition = DEPLOY = true
templating = spawn
updates = OBSTACLE_ALERT

// Parameters specific to this behavior
// ------------------------------------
allowable_ttc = 20 // default
pwt_outer_dist = 50 // default
pwt_inner_dist = 10 // default
completed_dist = 60 // default
min_util_cpa_dist = 8 // default
max_util_cpa_dist = 16 // default
use_refinery = true // default is false
visual_hints = obstacle_edge_color = white // default
visual_hints = obstacle_vertex_color = gray60 // default is white
visual_hints = obstacle_vertex_size = 1 // default is white
visual_hints = obstacle_fill_color = gray60 // default
visual_hints = obstacle_fill_transparency = 0.7 // default
visual_hints = buffer_min_edge_color = gray60 // default
visual_hints = buffer_min_vertex_color = blue // default is dodger_blue
visual_hints = buffer_min_vertex_size = 1 // default
visual_hints = buffer_min_fill_color = gray70 // default
visual_hints = buffer_min_fill_transparency = 0.25 // default
visual_hints = buffer_max_edge_color = gray60 // default
visual_hints = buffer_max_vertex_color = dodger_blue // default
visual_hints = buffer_max_vertex_size = 0 // default is 1
visual_hints = buffer_max_fill_color = gray70 // default
visual_hints = buffer_max_fill_transparency = 0.1 // default
}
```

### 13.4 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be suppressed or changed to a different variable name using the `post_mapping` configuration parameter described in Section 7.2.8.

- **AVD OB SPAWN:** A request to the obstacle manager specifying the conditions for obstacle alerts.
- **NOTED_RESOLVED**: A posting made when the behavior notes an obstacle has been reported be
  resolved by the obstacle manager.
- **VIEW_POLYGON**: A polygon rendering of either the obstacle, inner, or outer buffer.

### 13.5 Configuring and Using the AvoidObstacle Behavior

The AvoidObstacle behavior produces an objective function based on the relative position and
trajectory between the vehicle and its obstacle.

The objective function is based on applying a utility to the calculated closest point of approach
(CPA) for a candidate maneuver. The user may configure a priority weight, but this weight is
typically degraded as a function of the range to the obstacle. The behavior may be configured
for avoidance with respect to an obstacle known prior to the start of the mission, or it may be
configured to spawn a new instance upon demand as obstacles become known through the obstacle
manager.

### 13.6 Specifying the Behavior Priority Weight Policy

The AvoidObstacle behavior may be configured to increase its priority as it closes range to the
obstacle. The priority weight specified in its configuration represents the *maximum* possible priority
applied to the behavior, presumably in close range to the obstacle. The range at which this maximum
priority applies is specified in the `pwt_inner_dist` parameter. Likewise, the `pwt_outer_dist` parameter
specifies a range to the obstacle where the priority weight becomes zero, regardless of the priority
weight specified in the configuration file.

So the *current priority* will always be between zero and the maximum priority set in the behavior
`priority` configuration parameter. To be more precise:

Current Priority =

- 0 if current range to obstacle is greater than or equal to `pwt_outer_dist`
- 100 if current range to obstacle is less than or equal to `pwt_inner_dist`
- otherwise \((\frac{pwt\_outer\_dist - current\_range}{pwt\_outer\_dist - pwt\_inner\_dist}) \times priority\)

This relationship is shown in Figure 34.
Figure 34: Scaling priority weights based on ownship range to obstacle: The range between the vehicle and the obstacle affects whether the behavior is spawned, is active and with what priority weight. Beyond the range specified by $pwt_{\text{outer dist}}$, the behavior will have a zero priority weight, if it even exists. Within the range of $pwt_{\text{outer dist}}$, the behavior is active with a non-zero priority weight growing as the obstacle comes closer. Within the range of $pwt_{\text{inner dist}}$, the behavior is active with 100% of its configured priority weight.

The example shown below in Figure 34 shows the effect of the $pwt_{\text{outer dist}}$ parameter. The vehicle on the left is proceeding east, oblivious to the two approaching vessels. The two westbound vessels, $\text{ben}$ and $\text{cal}$ are simulated exactly on top of one another. They are oblivious to one another, but will use the collision avoidance behavior to avoid the eastbound vessel, $\text{abe}$. The only difference between $\text{ben}$ and $\text{cal}$ is that $\text{cal}$ begins winding up its priority weight at 80 meters range to $\text{abe}$, as opposed to 30 meters for $\text{ben}$. The short simulation shows the resulting difference in trajectory.
By default, the priority weight decreases linearly between the two depicted ranges. The \texttt{pwt\_grade} parameter allows the degradation from maximum priority to zero priority to fall more steeply by setting \texttt{pwt\_grade=quadratic}.

13.7 Specifying the Utility Policy of the Behavior

Whereas the behavior \textit{weight} discussed above in Section 24.5 determines the influence of the behavior relative to other behaviors, the behavior \textit{utility function} specifies the relative utility of candidate maneuvers from the perspective of the collision avoidance goals of this behavior.

The utility function for the avoid collision behavior is based on two factors:

- The range at the closest point of approach (CPA) for a candidate maneuver
- The determination risk associated with any CPA range.

The first component is just physics. Given ownship and contact current positions and trajectories, and the assumption that the contact will stay on its current heading and speed for at least the near future, the projected CPA range may be calculated for any given heading-speed maneuver. Consider the example in Figure 53 below with particular ownship and contact positions and trajectories.
shown on the left. On the right, the projected CPA range values are plotted for all candidate
ownership maneuvers.

The second component of the utility function is the mapping of "risk" to certain CPA range
values. This part is very subjective, and of course is why there are configuration parameters, allow

different users to align the behavior with their own notion of risk. The two key parameters are

\( \text{min}_\text{util}_\text{cpa}_\text{dist} \) and \( \text{max}_\text{util}_\text{cpa}_\text{dist} \). The former refers the the CPA range with the minimum
utility, essentially equivalent to a collision. For this value, perhaps pick a range that, while not
a collision, someone would get written up for a safety violation for breaching this range. The

\( \text{max}_\text{util}_\text{cpa} \) range, on the other hand, is the range that, above which there is not increase in utility.
Everything at this range or higher is "all clear".

These two parameters form a function, \( g() \), which takes as input the CPA range value and outputs
the utility of a candidate maneuver. This function is depicted on the left in Figure 54, and the
overall utility function is shown on the right:
Figure 37: (left) A utility function mapping CPA range values to a single risk utility is shown. Up to the range specified by the min_util_cpa configuration parameter, maneuvers resulting in these ranges are considered essentially collisions. Above the range specified by the max_util_cpa configuration parameter, maneuvers resulting in these ranges are considered essentially equivalently optimal. In between are the ranges that are in between disaster and optimal. These represent compromise maneuvers that the helm may opt for if there are other behaviors that find such maneuvers useful for accomplishing their objectives.

The min_util_cpa_dist and max_util_cpa_dist parameters have a default value of 10 and 75 meters respectively. These values are very subjective and will need to be adjusted per vehicle and mission. Currently these defaults are used if not specified by the user, but in future releases overriding these values may be enforced.
Figure 38: Two vehicles use the AvoidCollision behavior to avoid the oblivious and non-maneuvering eastbound vehicle. One vehicle, ben, is configured with the\texttt{min\_util\_cpa\_dist} and \texttt{max\_util\_cpa\_dist} parameters set to 5 and 15 meters respectively. The other vehicle, cal, is more risk averse and has these two parameters set to 15 and 25 meters respectively. The resulting trajectories of ben and cal are shown. The two vehicles on the right are simulated separately, unaware of each other, with the same starting position and parameters except for the two above parameters. The divergence in trajectory is solely due to the differences around these two parameters.

\texttt{video:(0:13): https://vimeo.com/458207817}
14 The PeriodicSurface Behavior

This behavior will periodically influence the depth and speed of the vehicle while remaining neutral at other times. The purpose is to bring the vehicle to the surface periodically to achieve some event specified by the user, typically the receipt of a GPS fix. Once this event is achieved, the behavior resets its internal clock to a given period length and will remain idle until a clock time-out occurs.

14.1 Configuration Parameters

*Listing 14.1: Configuration Parameters Common to All Behaviors.*

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the *active* state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the *idle* state. Section 7.2.6.
- **duration_reset**: A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the *idle* state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is not in the *active* state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
- **nostarve**: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.
- **perpetual**: If true allows the behavior to to run even after it has completed. Section 7.2.7.
- **post_mapping**: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- **priority**: The priority weight of the behavior. Section 7.2.3.
- **pwt**: Same as **priority**.
- **runflag**: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
- **spawnflag**: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.
- **templating**: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.
**updates**: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

*Listing 14.2: Configuration Parameters for the PeriodicSurface Behavior.*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acomms_mark_variable</td>
<td>The incoming MOOS variable for resetting the acomms period clock.</td>
</tr>
<tr>
<td>ascent_grade</td>
<td>Manner in which desired speed approaches zero on the approach to the surface. The default is <em>linear</em>. Section 14.2.6.</td>
</tr>
<tr>
<td>ascent_speed</td>
<td>Desired speed of the vehicle during the ascent mode. Section 14.2.5.</td>
</tr>
<tr>
<td>atsurface_status_var</td>
<td>MOOS variable indicating number of seconds at the surface. Section 14.2.4.</td>
</tr>
<tr>
<td>mark_variable</td>
<td>The incoming MOOS variable for resetting the period clock. Section 14.2.2.</td>
</tr>
<tr>
<td>max_time_at_surface</td>
<td>The maximum time, in seconds, the vehicle will wait at the surface. Section 14.2.8.</td>
</tr>
<tr>
<td>pending_status_var</td>
<td>MOOS variable written to with remaining time on idle clock. Section 14.2.3.</td>
</tr>
<tr>
<td>period</td>
<td>Duration of the WAITING mode. Section 14.2.1.</td>
</tr>
<tr>
<td>zero_speed_depth</td>
<td>The depth, in meters, at which the desired speed becomes zero on ascent. Section 14.2.7.</td>
</tr>
</tbody>
</table>

*Listing 14.3: Example Configuration Block.*

```plaintext
Behavior = BHV_PeriodicSurface
{
  // General Behavior Parameters
  // ---------------------------
  name       = periodic_surface   // example
  pwt        = 100                // default
  condition  = MODE==TRANSITING  // example
  updates    = PSURFACE_UPDATES   // example

  // Parameters specific to this behavior
  // ------------------------------------
  acomms_mark_variable = ACOMMS_RECEIVED  // example
  ascent_grade = linear               // default
  ascent_speed = -1                   // default
  atsurface_status_var = TIME_AT_SURFACE // default
  mark_variable = GPS_UPDATE_RECEIVED // default
  max_time_at_surface = 300            // default
  pending_status_var = PENDING_SURFACE // default
  period = TIME_AT_SURFACE            // default
  zero_speed_depth = 0                 // default
}
```
14.2 Detailed Discussions of Behavior Parameters

14.2.1 The period Parameter

The period parameter sets the duration of the period, in seconds, during which the behavior will remain in the IDLE_WAITING state.

14.2.2 The mark_variable Parameter

The mark_variable names a variable used for indicating when the behavior witnesses the event that would reset the period clock. On each iteration, the variable is checked against its last known value and if different, the clock is reset. The default value for this parameter is GPS_UPDATE_RECEIVED. If this variable is populated by another process with a value indicating the time a GPS fix is obtained, then the mark will occur on each GPS fix. Since the value of this argument names a MOOS variable, it is case sensitive.

14.2.3 The pending_status_var Parameter

The pending_status_var names a variable to be written to with the value of the remaining time on the idle clock, rounded to integer seconds. The default value is PENDING_SURFACE. Since the value of this argument names a MOOS variable, it is case sensitive.

14.2.4 The atsurface_status_var Parameter

The atsurface_status_var parameter names a variable to be written to with the number of seconds that the vehicle has been waiting at the surface (for the event indicated by the MOOS variable specified in the mark_variable parameter). The number of seconds is rounded to the nearest integer and will be zero when the vehicle is not at the surface. The default value is time_at_surface. Since the value of this parameter names a MOOS variable, it is case sensitive.

14.2.5 The ascent_speed Parameter

The ascent_speed parameter indicates the desired speed (m/s) of the vehicle during the ascent state. If left unspecified, the ascent speed will be equal to the current noted speed at moment it transitions into the ascent state.

14.2.6 The ascent_grade Parameter

The ascent_grade parameter indicates the manner in which the ascent speed approaches zero as the vehicle progresses toward the zero_speed_depth. It has four legal values: fullspeed, linear, quadratic, and quasi. The default is linear. In all four cases, the initial speed is determined by the parameter ascent_speed, and the desired speed will be zero once the zero_speed_depth has been achieved. The four settings determine the manner of slowing to zero speed during the ascent. The fullspeed setting indicates that desired speed should remain constant through the ascent right up to the instant the vehicle achieves zero_speed_depth. For the other three settings the speed reduction is relative to the starting depth (the depth noted at the outset of the ascent state) and the zero_speed_depth. With the linear setting, the speed reduction is linear. With the quadratic setting, the speed reduction is quadratic (quicker initial speed reduction). With the quasi setting
the speed reduction is between linear and quadratic. The value passed to this parameter is not case sensitive.

14.2.7 The zero_speed_depth Parameter

The zero_speed_depth parameter sets the depth (in meters) during the ascent state at which the desired speed becomes zero, and presumably further ascent is achieved through positive buoyancy.

14.2.8 The max_time_at_surface Parameter

The max_time_at_surface parameter sets the maximum time (in seconds) spent in the AT_SURFACE state, waiting for the event indicated by the mvar_variable, before the behavior transitions into the IDLE state.

14.3 Internal States of Periodic Surface Behavior

The behavior can be in one of four states as described in Figure 39 below.

![Figure 39: Possible modes of the PeriodicSurface behavior.](image)

In the WAITING state the behavior is simply waiting for its clock to wind down to zero. The duration is given by the period parameter listed below. The clock is active despite any other run conditions that may apply to the behavior. It is started when the behavior is first instantiated and also when the desired event occurs at the surface. The ASCENDING_BLOCKED state indicates that the behavior timer has reached zero, but another run condition has not been met. This is to prevent the behavior from trying to surface the vehicle when other circumstances override the need to surface. In the ASCENDING state, the behavior will produce an objective function over depth and speed to bring the vehicle to the surface. A couple parameters described below can determine the trajectory of the vehicle during ascent. This state can transition back to the ASCENDING_BLOCKED state if run conditions become no longer satisfied prior to the vehicle reaching the surface. In the AT_SURFACE state the vehicle is at the surface waiting for a specified event.
15 The ConstantDepth Behavior

This behavior will drive the vehicle at a specified depth. This behavior merely expresses a preference for a particular depth. If other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

15.1 Configuration Parameters

**Listing 15.1: Configuration Parameters Common to All Behaviors.**

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the *active* state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the *idle* state. Section 7.2.6.
- **duration_reset**: A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the *idle* state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is not in the *active* state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
- **nostarve**: Allows a behavior to assert a maximum staleness for a MOOS variable. Section 7.2.9.
- **perpetual**: If true allows the behavior to to run even after it has completed. Section 7.2.7.
- **post_mapping**: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- **priority**: The priority weight of the behavior. Section 7.2.3.
- **pwt**: Same as **priority**.
- **runflag**: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
- **spawnflag**: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.
- **templating**: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.
updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 15.2: Configuration Parameters for the ConstantDepth Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basewidth</td>
<td>The width of the base, in meters, in the produced ZAIC-style IvP function. Section 15.4.</td>
</tr>
<tr>
<td>depth</td>
<td>The desired depth of the vehicle, in meters.</td>
</tr>
<tr>
<td>duration</td>
<td>Behavior duration, in seconds. Mandatory configuration for this behavior. Section 15.3.</td>
</tr>
<tr>
<td>peakwidth</td>
<td>The width of the peak, in meters, in the produced ZAIC-style IvP function. Section 15.4.</td>
</tr>
<tr>
<td>summitdelta</td>
<td>The height of the summit delta parameter in the produced ZAIC-style IvP function. See Figure 22. Section 15.4.</td>
</tr>
</tbody>
</table>
| depth_mismatch_var | Name of the MOOS variable indicating the present delta between the desired depth and the current depth. Section 15.2. |}

Listing 15.3: Example Configuration Block.

```
Behavior = BHV_ConstantDepth
{
  // General Behavior Parameters
  // ---------------------------
  name   = const_dep_survey   // example
  pwt    = 100                // default
  condition = MODE==SURVEYING // example
  updates = CONST_DEP_UPDATES // example

  // Parameters specific to this behavior
  // ------------------------------------
  basewidth = 100             // default
  depth = 0                   // default
  depth_mismatch_var = DEPTH_DIFF // example
  duration = 0                // default
  peakwidth = 3               // default
  summitdelta = 50            // default
}
```

15.2 Variables Published

The behavior may be optionally configured to publish a variable indicating the discrepancy between the requested depth and the actual observed depth. The MOOS variable is named using the depth_mismatch_var parameter. This variable will only be published when the behavior is in the running state.
15.3 The duration Parameter

The duration parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to another behavior or event via condition variables, then setting duration=no-time-limit will result in no time duration checks for this behavior.

15.4 The ConstantDepth Objective Function

See Figure 22.

- basewidth: The width of the base, in meters in the produced objective function. The default is 100. See Figure 22 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.

- peakwidth: The width of the peak in meters in the produced objective function. The default is 3. See Figure 22 for more on the peak parameter used in the ZAIC tool for building IvP functions.

- summitdelta: The width of the base, in meters in the produced objective function. The default is 50. See Figure 22 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.
16  The ConstantHeading Behavior

This behavior will drive the vehicle at a specified heading. This behavior merely expresses a preference for a particular heading. If other behaviors also have a heading preference, coordination/compromise will take place through the multi-objective optimization process.

16.1  Configuration Parameters

Listing 16.1: Configuration Parameters Common to All Behaviors.

activeflag:  A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
condition:  Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
duration:  Time in behavior will remain running before declaring completion. Section 7.2.6.
duration_idle_decay:  When true, duration clock is running even when in the idle state. Section 7.2.6.
duration_reset:  A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
duration_status:  The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
endflag:  A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
idleflag:  A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
inactiveflag:  A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
name:  The (unique) name of the behavior. Section 7.2.2.
nostarve:  Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.
perpetual:  If true allows the behavior to run even after it has completed. Section 7.2.7.
post_mapping:  Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
priority:  The priority weight of the behavior. Section 7.2.3.
pwt:  Same as priority.
runflag:  A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
spawnflag:  A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.
templating:  Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.
updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 16.2: Configuration Parameters for the ConstantHeading Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basewidth</td>
<td>The width of the base, in degrees, in the produced ZAIC-style IvP function. Section 16.5.</td>
</tr>
<tr>
<td>complete_thresh</td>
<td>The threshold in discrepancy between the the requested heading and present heading before the behavior completes. The default is −1. Section 16.4.</td>
</tr>
<tr>
<td>duration</td>
<td>Behavior duration, in seconds. Mandatory configuration for this behavior. Section 16.3.</td>
</tr>
<tr>
<td>heading</td>
<td>The desired heading of the vehicle, in degrees (North=0).</td>
</tr>
<tr>
<td>peakwidth</td>
<td>The width of the peak, in degrees, in the produced ZAIC-style IvP function. Section 16.5.</td>
</tr>
<tr>
<td>summitdelta</td>
<td>The height of the summit delta parameter in the produced ZAIC-style IvP function. Section 16.5.</td>
</tr>
<tr>
<td>heading_mismatch_var</td>
<td>Name of the MOOS variable indicating the present delta between the desired heading and the current heading. Section 16.2.</td>
</tr>
</tbody>
</table>

Listing 16.3: Example Configuration Block.

```
Behavior = BHV_ConstantHeading
{
    // General Behavior Parameters
    // ---------------------------
    name = const_hdg // example
    pwt = 100 // default
    condition = MODE==GO_STRAIGHT // example
    updates = CONST_HDG_UPDATES // example
    
    // Parameters specific to this behavior
    // ------------------------------------
    basewidth = 10 // default
    duration = 0 // default
    speed = 0 // default
    heading_mismatch_var = HDG_DIFF // example
    peakwidth = 10 // default
    summitdelta = 25 // default
}
```

16.2 Variables Published

The behavior may be optionally configured to publish a variable indicating the discrepancy between the requested heading and the actual observed heading. The MOOS variable is named using the heading_mismatch_var parameter. This variable will only be published when the behavior is in the running state.
16.3 The duration Parameter

The duration parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to another behavior or event via condition variables, then setting duration = no-time-limit will result in no time duration checks for this behavior.

16.4 Behavior Completion

The behavior, by default, remains active so long as its conditions are met, just like any other behavior. If the user optionally declares a threshold via the complete_thresh parameter, the behavior will complete once the discrepancy between the observed heading and the goal heading falls below that threshold. In this case the behavior will complete, as any other helm behavior, by posting its endflags.

16.5 The ConstantHeading Objective Function

See Figure 22.

- **basewidth**: The width of the base, in degrees in the produced objective function. The default is 170. See Figure 22 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.

- **peakwidth**: The width of the peak in degrees in the produced objective function. The default is 10. See Figure 22 for more on the peak parameter used in the ZAIC tool for building IvP functions.

- **summitdelta**: The width of the base, in meters in the produced objective function. The default is 25. See Figure 22 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.


17 The ConstantSpeed Behavior

This behavior will drive the vehicle at a specified speed. This behavior merely expresses a preference for a particular speed. If other behaviors also have a speed preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

17.1 Configuration Parameters

Listing 17.1: Configuration Parameters Common to All Behaviors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activeflag</td>
<td>A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.</td>
</tr>
<tr>
<td>condition</td>
<td>Specifies a condition that must be met for the behavior to be running. Section 6.5.1.</td>
</tr>
<tr>
<td>duration</td>
<td>Time in behavior will remain running before declaring completion. Section 7.2.6.</td>
</tr>
<tr>
<td>duration_idle_decay</td>
<td>When true, duration clock is running even when in the idle state. Section 7.2.6.</td>
</tr>
<tr>
<td>duration_reset</td>
<td>A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.</td>
</tr>
<tr>
<td>duration_status</td>
<td>The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.</td>
</tr>
<tr>
<td>endflag</td>
<td>A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.</td>
</tr>
<tr>
<td>idleflag</td>
<td>A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.</td>
</tr>
<tr>
<td>name</td>
<td>The (unique) name of the behavior. Section 7.2.2.</td>
</tr>
<tr>
<td>nostarve</td>
<td>Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.</td>
</tr>
<tr>
<td>perpetual</td>
<td>If true allows the behavior to to run even after it has completed. Section 7.2.7.</td>
</tr>
<tr>
<td>post_mapping</td>
<td>Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.</td>
</tr>
<tr>
<td>priority</td>
<td>The priority weight of the behavior. Section 7.2.3.</td>
</tr>
<tr>
<td>pwt:</td>
<td>Same as priority.</td>
</tr>
<tr>
<td>runflag</td>
<td>A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.</td>
</tr>
<tr>
<td>spawnflag</td>
<td>A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.</td>
</tr>
<tr>
<td>templating</td>
<td>Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.</td>
</tr>
</tbody>
</table>
**updates:** A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

**Listing 17.2: Configuration Parameters for the ConstantSpeed Behavior.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basewidth</td>
<td>The width of the base, in meters per second, in the produced ZAIC-style IvP function. Section 17.4.</td>
</tr>
<tr>
<td>duration</td>
<td>Behavior duration, in seconds. Mandatory configuration for this behavior. Section 17.3.</td>
</tr>
<tr>
<td>peakwidth</td>
<td>The width of the peak, in meters per second, in the produced ZAIC-style IvP function. Section 17.4.</td>
</tr>
<tr>
<td>speed</td>
<td>The desired speed of the vehicle, in meters per second.</td>
</tr>
<tr>
<td>summitdelta</td>
<td>The height of the summit delta parameter in the produced ZAIC-style IvP function. Section 17.4.</td>
</tr>
<tr>
<td>speed_mismatch_var</td>
<td>Name of the MOOS variable indicating the present delta between the desired speed and the current speed. If left unspecified, no posting is made. Section 17.2.</td>
</tr>
</tbody>
</table>

**Listing 17.3: Example Configuration Block.**

```
Behavior = BHV_ConstantSpeed
{
    // General Behavior Parameters
    // ---------------------------
    name = const_spd_transit // example
    pwt = 100 // default
    condition = MODE==TRANSITING // example
    updates = CONST_SPD_UPDATES // example

    // Parameters specific to this behavior
    // ------------------------------------
    basewidth = 0.2 // default
    duration = 0 // default
    speed = 0 // default
    speed_mismatch_var = SPEED_DIFF // example
    peakwidth = 0 // default
    summitdelta = 0 // default
}
```

### 17.2 Variables Published

The behavior may be optionally configured to publish a variable indicating the discrepancy between the requested speed and the actual observed speed. The MOOS variable is named using the `speed_mismatch_var` parameter. This variable will only be published when the behavior is in the running state.
17.3  The duration Parameter

The duration parameter defined for all general behaviors, but for this behavior, specification is mandatory for safety reasons. The default if not specified is 0 seconds which will result in the behavior completing immediately. If no duration limit is desired, e.g., if the behavior is tied to another behavior or event via condition variables, then setting duration = no-time-limit will result in no time duration checks for this behavior.

17.4  The ConstantSpeed Objective Function

See Figure 22.

- peakwidth: The width of the peak in meters/second in the produced objective function. The default is 0. See Figure 22 for more on the peak parameter used in the ZAIC tool for building IvP functions.

- basewidth: The width of the base, in meters/second in the produced objective function. The default is 0.2. See Figure 22 for more on the basewidth parameter used in the ZAIC tool for building IvP functions.

- summitdelta: The width of the base, in meters/second in the produced objective function. The default is 0. See Figure 22 for more on the summitdelta parameter used in the ZAIC tool for building IvP functions.
18 The GoToDepth Behavior

This behavior will drive the vehicle to a sequence of specified depths and duration at each depth. This behavior merely expresses a preference for a particular depth. If other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process. A log of vehicle depth similar to figure below may result.

![Depth log from simulation](image)

Figure 40: Depth log from simulation with the depth parameters shown in Listing 3. The lighter, step-like line indicates the values of `DESIRED_DEPTH` generated by the helm, and the darker line indicates the recorded depth value of the vehicle. The depth plateaus start from the moment the vehicle achieves depth. For example, the vehicle achieved a depth of 45 meters at 119 seconds and retained that desired depth for another 60 seconds as requested in the configuration shown in Listing 3.

18.1 Configuration Parameters

*Listing 18.1: Configuration Parameters Common to All Behaviors.*

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the `active` state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the `idle` state. Section 7.2.6.
- **duration_reset**: A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the `idle` state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is not in the `active` state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
- **nostarve**: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.
perpetual: If true allows the behavior to run even after it has completed. Section 7.2.7.

post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.

priority: The priority weight of the behavior. Section 7.2.3.

pwt: Same as priority.

runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.

spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 18.2: Configuration Parameters for the GoToDepth Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capture_delta</td>
<td>The delta depth, in meters, between the current observed depth and the current target depth, below which the behavior will declare the depth to have been achieved. The default value is 1 meter. Section 18.2.2.</td>
</tr>
<tr>
<td>capture_flag</td>
<td>The name of a MOOS variable incremented each time a target depth level has been achieved. Section 18.2.1.</td>
</tr>
<tr>
<td>depth</td>
<td>A colon-separated list of comma-separated pairs. Each pair contains a desired depth and a duration at that depth. Section 18.2.3.</td>
</tr>
<tr>
<td>repeat</td>
<td>The number of times the vehicle will traverse through the evolution of depths. Section 18.2.4.</td>
</tr>
</tbody>
</table>

Listing 18.3: Example Configuration Block.

```plaintext
Behavior = BHV_GoToDepth
{
  // General Behavior Parameters
  // ---------------------------
  name        = gotodepth       // example
  pwt         = 100             // default
  condition   = MODE==Alpha    // example
  updates     = GOTO_DEPTH_UPDATES // example

  // Parameters specific to this behavior
  // ------------------------------------
  capture_delta = 1               // default (meters)
  capture_flag  = DEPTH_ACHIEVED  // example
  depth = 40,60:30,45:20,45       // example
  repeat = 0                      // default
}
```
18.2 A Detailed Discussion of GoToDepth Behavior Parameters

18.2.1 The capture_flag Parameter

This parameter names a MOOS variable incremented each time a target depth level has been achieved. It may be useful for logfile analysis and also allows other behaviors to be conditioned on a depth event. If this behavior is completed in perpetual mode, the counter is reset to zero. If the behavior is repeating a set of depths by setting repeat greater than zero, the counter will continue to increment through evolutions. The default value is the empty string, meaning nothing will be posted. Note the named MOOS variable will automatically have the prefix "GTD." applied.

18.2.2 The capture_delta Parameter

When the GoToDepth behavior is running and actively influencing the depth of the vehicle to a target depth level, it monitors the discrepancy between the observed depth and the current target depth. The capture_delta parameter is the delta depth, in meters, below which the behavior will declare the depth to have been achieved. The default value is 1 meter.

As an example, consider a target depth of say 100 meters, and a capture_delta of 5. When a diving vehicle reaches 95 meters, it will consider the depth achieved. However, it will continue to influence the vehicle to a depth of 100 meters for as long as prescribed by the depth setting. The "achieving" of the depth results in two things: (a) the duration clock for time spent at that depth will begin, and (b) if there is a capture_flag set, this flag will be posted by the helm to the MOOSDB.

18.2.3 The depth Parameter

The depth parameter is a colon-separated list of comma-separated pairs. Each pair contains a desired depth and a duration at that depth. The duration applies from the point in time that the depth is first achieved. If a time duration is not provided for any pair, it defaults to zero. Thus depth=20 is a valid parameter setting.

The duration is specified in seconds and reflects the time at depth after the vehicle has first achieved that depth, where achieving depth is defined by the capture_delta parameter. The behavior subscribes for NAV_DEPTH to examine the current vehicle depth against the target depth. If the current depth is within the delta given by capture_delta, that depth is considered to have been achieved. The behavior also stores the previous depth from the prior behavior iteration, and if the target depth is between the prior depth and current depth, the depth is considered to be achieved regardless of whether the prior or current depth is actually within the capture_delta.

18.2.4 The repeat Parameter

The number of times the vehicle will traverse through the evolution of depths, proceeding to the 1st depth after the nth depth has been hit. The default value is zero.

18.2.5 The perpetual Parameter

The perpetual parameter is defined at the superclass level, but it’s worth discussing its function here. If equal to true, when the vehicle completes its evolution of depths (perhaps several evolutions
if \texttt{repeat} is non-zero), the endflags will be posted. But rather than setting the complete variable to true and thus never receiving any further run consideration, the behavior is reset to its initial state. Presumably the user sets endflags that will cause the condition flags to be not immediately satisfied, thus putting the behavior in a state waiting again for an external event flag to be posted. The default value of this parameter is false.
19 The MemoryTurnLimit Behavior

The objective of the Memory-Turn-Limit behavior is to avoid vehicle turns that may cross back on its own path and risk damage to the towed equipment. Its configuration is determined by the two parameters described below which combine to set a vehicle turn radius limit. However, it is not strictly described by a limited turn radius; it stores a time-stamped history of recent recorded headings and maintains a heading average, and forms its objective function on a range deviation from that average. This behavior merely expresses a preference for a particular heading. If other behaviors also have a heading preference, coordination/compromise will take place through the multi-objective optimization process. The following parameters are defined for this behavior:

19.1 Configuration Parameters

Listing 19.1: Configuration Parameters Common to All Behaviors.

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the idle state. Section 7.2.6.
- **duration_reset**: A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
- **nostarve**: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.
- **perpetual**: If true allows the behavior to run even after it has completed. Section 7.2.7.
- **post_mapping**: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- **priority**: The priority weight of the behavior. Section 7.2.3.
- **pwt**: Same as priority.
- **runflag**: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 19.2: Configuration Parameters for the MemoryTurnLimit Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_time</td>
<td>The duration of time for which the heading history is maintained and heading average calculated. The default value is -1, indicating that the parameter is un-set. In this case the behavior will not produce an objective function.</td>
</tr>
<tr>
<td>turn_range</td>
<td>The range of heading values deviating from the current heading average outside of which the behavior reflects sharp penalty in its objective function. The default value is -1, indicating that the parameter is un-set. In this case the behavior will not produce an objective function.</td>
</tr>
</tbody>
</table>

Listing 19.3: Example Configuration Block.

```
Behavior = BHV_MemoryTurnLimit
{
  // General Behavior Parameters
  // ---------------------------
  // ---------------------------
  name = mem_turn_limit       // default
  pwt  = 100                  // default
  condition = MODE=TRANSITING // example
  condition = ARRAY=connected // example
  updates = MEM_TURN_UPDATES  // example

  // Parameters specific to this behavior
  // ------------------------------------
  memory_time = 60            // example (seconds)
  turn_range = 30             // example (degrees)
}
```

19.2 Calculation of the Heading History

The heading history is maintained locally in the behavior by storing the currently observed heading and keeping a queue of n recent headings within the memory_time threshold. The heading average calculation below handles the issue of angle wrap in a set of n headings $h_0 \ldots h_{n-1}$ where each heading is in the range $[0, 359]$.

$$\text{heading} = \text{atan2}(s, c) \cdot 180/\pi,$$

where $s$ and $c$ are given by:

161
\[
s = \sum_{k=0}^{n-1} \sin (h_k \pi/180)), \quad c = \sum_{k=0}^{n-1} \cos (h_k \pi/180)).
\]

The vehicle turn radius \( r \) is not explicitly a parameter of the behavior, but is given by:

\[
r = \frac{v}{(u/180) \pi),
\]

where \( v \) is the vehicle speed and \( u \) is the turn rate given by:

\[
u = \frac{\text{turn range}}{\text{memory time}}.
\]

The same turn radius is possible with different pairs of values for \text{turn range} and \text{memory time}. However, larger values of \text{turn range} allow sharper initial turns but temper the turn rate after the initial sharper turn has been achieved.

19.3 Variables Published

The only variable published by this behavior is \text{MEM\_TURN\_AVG} which indicates the present heading average, rounded to the nearest integer.

19.4 The MemoryTurnLimit Objective Function

A Rendering of the MemoryTurnLimit Objective Function

Figure 41: The MemoryTurnLimit objective function: The objective function produced by the MemoryTurnLimit behavior is defined over possible heading values. Depicted here is an objective function formed when the recent heading history is 225 degrees and the \text{turn range} parameter is set to 30 degrees. The resulting objective function highly favors headings in the range of 190-240 degrees. One the right is a "birds-eye" view of the function, and on the right the function is viewed at an angle to appreciate the 3D quality of the function. Higher (red) values correspond to higher utility.
20 The StationKeep Behavior

This behavior is designed to keep the vehicle at a given lat/lon or x,y station-keep position by varying the speed to the station point as a linear function of its distance to the point. The parameters allow one to choose the two distances between which the speed varies linearly, the range of linear speeds, and a default transit speed if the vehicle is outside the outer radius.

![Diagram of StationKeep behavior parameters]

Figure 42: The station-keep behavior parameters: The station-keep behavior can be configured to approach the outer station circle with a given transit speed, and will decrease its preference for speed linearly between the outer radius and inner radius. The preferred speed is zero when the vehicle is at or inside the inner radius.

An alternative to this station keeping behavior is an active loiter around a very tight polygon with the Loiter behavior. This station keeping behavior conserves energy and aims to minimize propulsor use. The behavior can be configured to station-keep at a pre-set point, or wherever the vehicle happens to be when the behavior transitions into an active state.

The station-keep behavior was initially developed for use on an autonomous kayak. It’s worth pointing out that a vehicle’s control system, i.e., the front-seat driver described in Section 2.3, may have a native station-keeping mode, in which case the activation of this behavior would be replaced by a message from the backseat autonomy system to invoke the station-keeping mode. It’s also worth pointing out that most UUVs are positively buoyant and will simply come to the surface if commanded with a zero-speed.

20.1 Configuration Parameters

Listing 20.1: Configuration Parameters Common to All Behaviors.

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
duration: Time in behavior will remain running before declaring completion. Section 7.2.6.
duration_idle_decay: When true, duration clock is running even when in the idle state. Section 7.2.6.
duration_reset: A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
duration_status: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
endflag: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
idleflag: A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
inactiveflag: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
name: The (unique) name of the behavior. Section 7.2.2.
nostarve: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.
perpetual: If true allows the behavior to run even after it has completed. Section 7.2.7.
post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
priority: The priority weight of the behavior. Section 7.2.3.
pwt: Same as priority.
runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.
templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.
updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 20.2: Configuration Parameters for the StationKeep Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>center_activate</td>
<td>If true, station-keep at the vehicle’s present position upon activation. Section 20.2.</td>
</tr>
<tr>
<td>hibernation_radius</td>
<td>A radius used for low-power, passive station-keeping. Section 20.5.</td>
</tr>
<tr>
<td>inner_radius</td>
<td>Distance to station-point within which the preferred speed is zero. Section 20.4.</td>
</tr>
<tr>
<td>outer_radius</td>
<td>Distance within which the preferred speed begins to decrease. Section 20.4.</td>
</tr>
<tr>
<td>outer_speed</td>
<td>Preferred speed at outer radius, decreasing toward inner radius. Section 20.4.</td>
</tr>
<tr>
<td>station_pt</td>
<td>An x,y pair given as a point in local coordinates. Section 20.2.</td>
</tr>
<tr>
<td>swing_time</td>
<td>Duration of drift of station circle with vehicle upon activation. Section 20.3.</td>
</tr>
<tr>
<td>transit_speed</td>
<td>Preferred speed beyond the outer radius. Section 20.4.</td>
</tr>
</tbody>
</table>
visual_hints: Preferences for rendering visual artifacts produced by the behavior. Section 20.7.

Listing 20.3: Example Configuration Block.

```c
Behavior = BHV_StationKeep
{

  // General Behavior Parameters
  // ---------------------------
  name = station-keep // example
  pwt = 100 // default
  condition = MODE==SKEEPING // example
  inactiveflag = STATIONING = false // example
  activeflag = STATIONING = true // example

  // Parameters specific to this behavior
  // ------------------------------------
  center_activate = false // default
  hibernation_radius = -1 // default
  inner_radius = 4 // default
  outer_radius = 15 // default
  outer_speed = 1.2 // default
  transit_speed = 2.5 // default
  station_pt = 0,0 // default
  swing_time = 0 // default

  visual_hints = vertex_size = 1 // default
  visual_hints = edge_color = light_blue // default
  visual_hints = edge_size = 1 // default
  visual_hints = label_color = white // default
  visual_hints = vertex_color = red // default

}
```

20.2 The station_pt and center_activate Parameters

The station-keep point is set in one of two ways: either with a pre-specified fixed position, or with the vehicle’s current position when the vehicle transitions into the running state. To set a fixed station-keep position:

```
station_pt = 100,250
```

To configure the behavior to station-keep at the vehicle’s current position when it enters the running state:

```
center_activate = true // "true" is case insensitive
```

20.3 The swing_time Parameter

At the outset of station-keeping via center_activate, the vehicle typically is moving at some speed. Despite the fact that station-keeping is immediately active and typically results in a desired speed of
zero if no other behaviors are active, the vehicle will continue some distance before coming to a near or complete stop in the water, thus "over-shooting" the station-keep point. This often means that the station-keep behavior will immediately turn the vehicle around to come back to the station-keep point. This can be countered by setting the behavior’s `swing_time` parameter, the amount of time after initial center-activation that the station-keep point is allowed to drift with the current position of the vehicle before becoming fixed. The format is:

\[
\text{swing\_time} = \text{<time-duration>} \quad \text{// default is 0 seconds}
\]

The time duration is given in seconds and should be in the range [0, 60]. If found to be outside this range it is simply clipped to the boundary value.

If the behavior enters the running state, but center-activation is not set to true, and no pre-specified fixed position is given, the behavior will not produce an objective function. It will remain in the running state, but not the active state. (Section 6.5.3 discusses run states.) In this situation, a warning will be posted via the helm appcast structure and by posting to the `BHV\_WARNING` variable. In both cases the warning will read: "STATION\_POINT\_NOT\_SET".

### 20.4 The `inner_radius`, `outer_radius`, and `outer_speed` Parameters

The `inner_radius` and `outer_radius` parameters affect the preferred speed of the behavior as it relates to the vehicle’s current range to the station point. The preferred speed at the outer radius is given by the parameter `outer_speed`. The preferred speed decreases linearly to zero as the vehicle approaches the inner radius. The default values for the inner and outer radii are 4 and 15 respectively. If configured with values such that the inner is greater than the outer, this will not trigger an error, but the two radii parameters will be collapsed to the value of the inner radius on the first iteration of the behavior. The `transit_speed` parameter indicates the desired speed when the vehicle is outside the `outer_radius`. The default value for `transit_speed` is 2.5 meters per second. If the `outer_speed` is set higher than the `transit_speed` the `transit_speed` will automatically be raised to the `outer_speed`.

### 20.5 Passive Low-Energy Station Keeping Mode

The station-keep behavior can be configured to operate in a "passive" mode. This mode differs from the default mode primarily in the way it acts after it reaches the inner-radius, i.e., the point at which the behavior regards the vehicle to be on-station and outputs a preferred speed of zero. In the normal mode, the behavior will begin to output a preferred heading and non-zero speed as soon as the vehicle slips beyond the inner-radius. In the passive mode, the behavior will let the vehicle drift or otherwise move to a distance specified by the `hibernation_radius` before it resumes outputting a preferred heading and non-zero speed. The idea is shown in Figure 43.
Figure 43: **Passive station-keeping:** The station-keep behavior can be configured in the "passive" mode. The vehicle will move toward the station point until it reaches the inner_radius or until progress ceases. It will then drift until its distance to the station point is beyond the hibernation_radius. At this point it will re-engage to reach the station-point and may trigger another behavior to dive.

This mode was built with UUVs in mind. Most UUVs are deployed having a positive buoyancy (battery dies - vehicle floats to the surface). They need to be moving at some speed to maintain a depth. Furthermore, it may not be safe to assume that a UUV can effectively execute a desired heading when it is operating on the surface. For these reasons, when operating in the passive mode, this behavior will publish a variable indicating whether it is in the mode of drifting or attempting to make progress toward the station point. The status is published in the variable PSKEEP_MODE, short for "passive station-keeping mode". This variable will be set to "SEEKING_STATION" when outputting a non-zero speed preference, and presumably moving toward the station-point. The variable will be set to "HIBERNATING" otherwise. This opens the option of configuring the helm with the ConstantDepth behavior to work in conjunction with the StationKeep behavior by conditioning the ConstantDepth behavior to be running only when PSKEEP_MODE="SEEKING_STATION". The idea is shown in Figure 44.
This behavior mode is regarded as "low-power" due to the presumably long periods of drifting before resuming actively seeking the station point. A couple of safeguards are designed to ensure that when the behavior is in the "STATIONSEEKING" mode, that it does not get hung or stuck in this mode for much longer than intended or needed. How could one become stuck in this mode? Two ways - by either reaching an equilibrium at-speed, (and perhaps at-depth) state where the vehicle is neither progressing toward or way from the inner_radius, or by repeatedly "missing" the inner_radius by heading right past it.

Both cases can be guarded against and detected by monitoring the history of vehicle speed in the direction of the station-point. If this speed becomes zero, an equilibrium state is assumed, and if it becomes negative, it is assumed that the vehicle missed the inner radius circle entirely. In short, the StationKeep behavior exits the "STATIONSEEKING" mode and enters the "HIBERNATING" mode when it detects the vehicle speed toward the station-point reach zero. To calculate this vehicle speed, a ten-second history of range to the station-point is kept by the behavior. A zero speed, or "stale-progress" criteria is declared simply if the range to the station-point for the most recent measure in the history list is not less than the range of ten seconds ago in the history list. The behavior will transition into the "HIBERNATING" mode if either the inner-radius or stale-progress criteria are met.

It is also possible that when the StationKeep behavior enters the "SEEKING_STATION" mode from the "HIBERNATING" mode, that the vehicle initially begins to open its range to the station-point before it begins to close range. This would be expected, for example, if the vehicle were pointed away from the station-point when the behavior first entered the "SEEKING_STATION" mode. In this case it’s quite possible that the behavior would correctly, but unwantingly, infer that the stale-progress criteria has been met. For this reason, the stale-progress criteria is not applied until an "initial-progress" criteria is met after entering the "SEEKING_STATION" mode. The same ten second history is used to detect when the vehicle begins to make initial progress, i.e., closing range, toward the station-point.

20.6 Station Keeping On Demand

A common, and perhaps recommended configuration, is to have one station-keep behavior defined for a given helm configuration and have it set to be usable in one of three ways: (a) station-keep at a default pre-specified position, (b) station-keep at a specified position dynamically provided, or (c)
station-keep at the vehicle’s present position when activated. The behavior would be configured as follows:

```
station_pt = 100,200 // The default station-keep point
center_activate = false
updates = STATION_UPDATES
condition = STATION_REQUEST = true
```

Then, to use the station-keep behavior in the above three ways, the following three pairs of postings, i.e., pokes, to the MOOSDB would be used. See Section 7.2.5 for more on the `updates` parameter defined for all behaviors - by utilizing this dynamic configuration hook, the one behavior configuration above can be used in these different manners. The first pair would result in the behavior keeping station at its pre-arranged point of (100,200):

```
STATION_REQUEST = true
STATION_UPDATES = center_activate=false
```

The second line above dynamically configures the behavior parameter `center_activate` to be false to ensure that the point given by the original `station_pt` parameter is used. Even though the `center_activate` parameter is initially set to false, the above usage sets it to false anyway, to be safe, and in case it has been dynamically set to true in a prior usage.

In the second case below, again the `center_activate` parameter is dynamically set to false for the same reasons. In this case the `station_point` parameter is also dynamically configured with a given point:

```
STATION_REQUEST = true
STATION_UPDATES = station_pt=45,-150 # center_activate=false
```

In the last case, below, the behavior is activated and configured to station-keep at the vehicle’s present position when activated. There is no need to tinker with the `station_pt` parameter since this parameter is ignored when `center_activate` is true:

```
STATION_REQUEST = true
STATION_UPDATES = center_activate=true
```

It’s worth noting that above variable-value pairs that trigger the StationKeep behavior could have come from a variety of sources. They could be endflags from another behavior. They could have come from a poke using `uPokeDB`, `uTimerScript`, `pMarineViewer` or any third party command and control interface.

### 20.7 The `visual_hints` Parameter

Although the primary output of the StationKeep behavior is an IvP Function, a number of visual properties are also published for convenience in mission monitoring. This includes (a) the `inner_radius`, (b) the `outer_radius`, and (c) the `hibernation_radius` if used. These visual artifacts
have default properties in size and color that may be altered to the user’s preferences. These preferences are configurable through the `visual_hints` parameter. Each parameter below is used in the following way by example:

| visual_hint = vertex_size=3, edge_size=2 |
| visual_hint = vertex_color=khaki |

- **edge_color**: The color of edges rendered in the loiter polygon. The default is "white".
- **edge_size**: The width of edges rendered in the loiter polygon. The default is 1.
- **label_color**: The color of labels rendered with the inner and outer radii. The default is "gray50".
- **vertex_color**: The color of vertices rendered in the loiter polygon. The default is "dodger_blue".
- **vertex_size**: The size of vertices rendered in the loiter polygon. The default is 1.

Rendering of vertices may be shut off with a size of zero, and labels may be shut off with the special color "invisible". For a list of legal colors, see Appendix C.
21 The Timer Behavior

The Timer behavior is a somewhat unique behavior in that it never produces an objective function. It has virtually no functionality beyond what is derived from the parent IvPBehavior class. It can be used to set a timer between the observation of one or more events (with condition flags) and the posting of one or more events (with end flags). The duration, duration_status, duration_idle_decay, condition, runflag and endflag parameters are all defined generally for behaviors. There are no additional parameters defined for this behavior.

21.1 Configuration Parameters

Listing 21.1: Configuration Parameters Common to All Behaviors.

- activeflag: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- condition: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- duration: Time in behavior will remain running before declaring completion. Section 7.2.6.
- duration_idle_decay: When true, duration clock is running even when in the idle state. Section 7.2.6.
- duration_reset: A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
- duration_status: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- endflag: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- idleflag: A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
- inactiveflag: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
- name: The (unique) name of the behavior. Section 7.2.2.
- nostarve: Allows a behavior to assert a maximum staleness for a MOOS variable. Section 7.2.9.
- perpetual: If true allows the behavior to run even after it has completed. Section 7.2.7.
- post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- priority: The priority weight of the behavior. Section 7.2.3.
- pwt: Same as priority.
- runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
- spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.
**templating:** Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

**updates:** A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 21.2: Example Configuration Block.

```plaintext
Behavior = BHV_Timer
{
    // General Behavior Parameters
    // ------------------------------------
    name = bhv_timer // default
    condition = WAIT_REQUEST=true // example
    duration = 120 // example
    idleflag = WAITING=false // example
    runflag = WAITING=true // example
    runflag = WAITED=false // example
    endflag = WAITED=true // example
    updates = TIMER_UPDATES // example

    // Parameters specific to this behavior
    // ------------------------------------
    // None

    }
}
```

### 21.2 Variables Published

This behavior publishes two variables for monitoring and logging performance - `TIMER_IDLE, TIMER_RUNNING`. 
22 The TestFailure Behavior

The TestFailure behavior is used to test the helm in two conceivable behavior failure modes. First, it may be used to simulate a behavior that crashes and thereby results in the crash of the helm. Second, it may be used to simulate a behavior that consumes a sufficiently large enough amount of time so as cause the helm to be considered "hung" by consumers of the helm output.

Recall that the helm is compiled, with behaviors, into a single MOOS application. Although some behaviors may be compiled into shared libraries loaded at run time, thereby not requiring a recompile, all behaviors do run as part of a single helm process. A crashed behavior results in a crashed helm. Furthermore the helm, on each iteration, queries each participating behavior for its input. It does not do this in separate threads, and there is no timeout with a default reply should a behavior never answer. A hung behavior results in a hung helm. These are architecture decisions that on one hand allow a substantial amount of simplicity in the helm implementation and debugging. Furthermore, it’s not clear that a graceful and safe policy exists to safely handle a rogue behavior other than to either (a) abort the mission or (b) put the vehicle in the hands of a much more conservatively configured "standby" instance of the helm, perhaps just to get the vehicle home. This behavior is used to simulate both kinds of rogue behaviors, a behavior that crashes and a behavior the hangs. The crash is implemented simply with an assert(0) statement, and the hang is implemented with a long for-loop.

22.1 Configuration Parameters

Listing 22.1: Configuration Parameters Common to All Behaviors.

- **activeflag**: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- **condition**: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- **duration**: Time in behavior will remain running before declaring completion. Section 7.2.6.
- **duration_idle_decay**: When true, duration clock is running even when in the idle state. Section 7.2.6.
- **duration_reset**: A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
- **duration_status**: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- **endflag**: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- **idleflag**: A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
- **inactiveflag**: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
- **name**: The (unique) name of the behavior. Section 7.2.2.
nostarve: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.

perpetual: If true allows the behavior to to run even after it has completed. Section 7.2.7.

post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.

priority: The priority weight of the behavior. Section 7.2.3.

pwt: Same as priority.

runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.

spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 22.2: Configuration Parameters for the TestFail Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>failure_type</td>
<td>Fail by crash, or fail by hang.</td>
</tr>
</tbody>
</table>

Choosing to fail with a crash will result in behavior executing an assert(0) as soon as the behavior enters the running state. Configuring for a hang will likewise result in the execution of a long for-loop upon entering the running state. The hang time is by default three seconds, but may be altered by specifying a time alongside the hang parameter as in line 9 below.

Below are a couple typical usage configurations. In the first case, the TestFailure behavior is set with a condition as on line 4, to remain idle until the vehicle is deployed. The duration parameter on line 6, defined for all helm behaviors, ensures the behavior will not run until two minutes into the mission. This is done presumably to have the failure occur while the vehicle is "in the middle of doing something". Setting the duration_idle_decay parameter to false indicates the duration countdown is not to proceed while the behavior is still idle. In this case the failure type is set to hang for two seconds. This behavior configuration is used in the Kilo example mission, described in Section XYZ, and can be run to illustrate.

Listing 22.3: Example Configuration Block.
Another configuration style below will result in the behavior crashing as soon as the behavior meets its run condition. Presumably the user can simply poke the MOOSDB at any time to invoke the failure, or another MOOS app may generate the poke at the key desired moment. The `failure.type` is left unspecified since `crash` is the default failure type.

Listing 22.4: Example Configuration Block.
23 Contact Related Behaviors of the IvP Helm

The section contains a description of five behaviors currently written for the helm that reason about relative position to another vehicle or contact.

- The AvoidCollision behavior
- The AvdColregsV19 behavior
- The CutRange behavior
- The Shadow behavior
- The Trail behavior

Each behavior needs to be continuously updated with the position and trajectory of a given contact. The helm subscribes to the MOOS variable, NODE_REPORT, which may have content similar to:

```
NODE_REPORT= "NAME=alpha,TYPE=UUV,MOOSDB_TIME=39.01,TIME=1252348077.59,
X=51.71,Y=-35.50,LAT=43.824981,LOM=-70.329755,SPD=2.00,HDG=118.85,
YAW=118.84754,DEPTH=4.63,LENGTH=3.8,MODE=SURVEYING"
```

This contact information is stored in the helm information buffer for any behavior that wishes to retrieve it on any iteration. Since each behavior type needs to reason about contact information, the common code for this is handled in superclass called IvPContactBehavior. So for these behaviors, the class hierarchy looks like:

![Class Hierarchy Diagram]

Figure 45: IvP Contact Behaviors: The contact-related behaviors share a common superclass to streamline the development of contact behaviors and provide a level consistency between behaviors.

23.1 Static Versus Dynamically Spawned Behaviors

A contact behavior may be configured in one of two ways. The simplest way is to reason about a particular contact with a name known at mission time. This is more appropriate for a behavior that needs to operate on one contact at a time, such as a trail or cut-range behavior. While it is technically possible to try to cut-range or trail multiple contacts simultaneously, and the IvP solver will support such an attempt, typically such behaviors are focused on a single contact. Configuration of the behavior simply names the contact

```
contact = mothership
```
A second type of configuration is possible when we would like to have several simultaneous instances of the same behavior, as during collision avoidance with multiple contacts. In this case the behavior is configured to enable *templating*, and the contact name is left unspecified until a new contact emerges:

\[
\text{templating} = \text{spawn} \\
\text{contact} = \text{to}\_\text{be}\_\text{determined}
\]

A templating contact behavior works closely with a contact manager, either the pBasicContactMgr app or the newer version, pContactMgrV20. The contact manager is responsible for generating the MOOS posting events that trigger the spawning of a new behavior. As depicted in Figure 46 below, the contact manager applies a certain configurable criteria to a contact, and if it passes that criteria, it generates an alert to the helm, resulting in a spawned behavior.

![Relationship between the contact manager and spawned contact-related behaviors](image)

**Figure 46:** Relationship between the contact manager and spawned contact-related behaviors: (1) A contact closes range to ownship, crossing a range threshold. (2) The contact manager makes note and changes the internal record associated with the contact. (3) An alert it generated and received by the helm. (4) The helm spawns a new contact-related behavior dedicated to this contact.

Typically the contact manager criteria for generating an alert is based on the *range* of the contact to ownship. However, the contact manager may also utilize additional filters, providing user configurable selectivity as to which contacts generate alerts resulting in behavior spawnings. These filters may be configured at the contact manager level, e.g., ignore all contacts of type sailboat. Or they may be configured at the behavior level, e.g., one collision avoidance behavior matches sailboats, and another collision avoidance behavior matches motorboats. This is covered in part in the documentation of pContactMgrV20, but portion that is accessible through behavior configuration is described next.

### 23.2 Exclusion Filters

An *exclusion filter* is a tool for allowing the contact manager and/or the helm to treat distinct contacts differently, or ignore selectively handle contacts with a certain property. The contact manager has an exclusion filter, and all behaviors have an exclusion filter. They can safely ignored if one wishes to simple treat all contacts equally, but they can provide some powerful options when applied to particular situation.
23.2.1 Types of Contact Exclusion Filters

Contact exclusion filters may be configured around four different types of properties:

- **name**: This is the most direct method but rather brittle. But if there is a particular named vehicle that you’d like to handle differently, it can be just called out by name.
- **type**: The vehicle type is part of the incoming NODE_REPORT message, e.g., UUV, USV and so on.
- **group**: The vehicle group is part of the incoming NODE_REPORT message, e.g., friendly, foe and so on.
- **region**: A convex polygon describing a region of the operation area.

Each of the above four properties can be configured as match or ignore filters. For example, `match_type=kayak` indicates that the contact must by of type kayak. Likewise `ignore_type=kayak` indicates that the contact must be of some type other than kayak. The same is true for `name`, `group`, and `region` properties. This makes eight possible configuration parameters available to all contact behaviors when templating is enabled:

- match_group and ignore_group
- match_name and ignore_name
- match_type and ignore_type
- match_region and ignore_region

Multiple entries for each property may be used. If for example multiple `match_name` configurations are used, a contact will pass the filter if it matches at least one. If multiple `ignore_name` configurations are used, a contact will pass the filter only if it has a name different from all.

The `region` property is a bit different from the other three properties. While a contact’s `name`, `type`, and `group` properties tend not to change as a mission unfolds, the contact position does. When the `region` is used by the contact manager as a filter it applies only to that moment in time. If, for example, the contact was outside a `match_region` but then enters it, it will now pass the filter and an alert will be generated, provided all other filter and range criteria are met. Likewise if the contact manager generated an alert for a contact, but then moved out of a `match_region` or into an `ignore_region`, the previously generated alert will not somehow be retracted.

One last note about `region` filters: non-convex regions may be implemented by using two or more convex regions that "cover" the desired non-convex region. And consistent with the above discussion, the approximating convex regions may harmlessly overlap.

23.2.2 Configuring Exclusion Filters Globally or Locally

As shown in Figure 46 above, the contact manager is the gatekeeper for sending the helm alert messages that may trigger the spawning of the behavior. Users can configure the exclusion filters in one of two ways:

- Global configuration via the contact manager, pContactMgrV20, or
- Local configuration via individual behaviors in the helm.
A third configuration method, dealing with behaviors after they have been spawned, is discussed in Section ??.

The former is method is arguably more convenient but the filters apply to all behavior templates equally. If the contact manager is configured to ignore sailboats, there is no way to otherwise configure the helm to have a spawnable behavior that will handle sailboats. Maybe you really do want to just ignore all sailboats. But if instead you’d like to have one type of behavior ignore sailboats and another type of behavior ignore motorboats, then the filter needs to be configured at the behavior level instead.

Figure 47 below depicts a helm locally configured with two distinct behavior templates, with the filter configuration done in the helm configuration. In this way, one distinct behavior is used for Contact Abe, and another for Contact Ben. It may be that both are collision avoidance behaviors, but perhaps the safety region around Abe is larger than Ben. Likewise the filter could have involved the vehicle type, group or region.

![Figure 47: Configuration of contact filter at the local behavior level.](image)

Using both global and local exclusion filters is certainly allowable, but keep in mind that if a contact is excluded at the global contact manager level, it does not matter what configurations are specified at the local behavior level.

### 23.2.3 Enabling Strict Filtering

Filtering based on the contact type or group depends on this information being known about the contacts. Typically this is embedded in each incoming NODE_REPORT. In some cases, especially in sensor-based contact management, this information may not be known.

For match filtering, this is not an issue. This is because the spirit of match filtering is that, for example if we say that a contact must be of type motor-boat, and the contact type is unknown, it’s pretty clear that condition is not satisfied and thus the contact with unknown type is filtered out.

With ignore filtering it is more ambiguous. This is because the spirit of ignore filtering is that, for example if we say that a contact cannot be of type motor-boat, and the contact type is unknown,
we might be inclined to say that it should not be filtered out. The user may however want to be strict about this and say that it should be filtered out since we don’t know the contact type, and for all we know it could be a motor-boat.

To allow the user to have some control and be explicit about the above situation, filtering can be configured to be strict by setting strict_ignore to be true or false. The below set of examples hopefully makes this clear.

<table>
<thead>
<tr>
<th>Contact Type</th>
<th>Ignore Group</th>
<th>Strict Ignore</th>
<th>Filtered Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>sailboat</td>
<td>sailboat</td>
<td>true or false</td>
<td>yes</td>
</tr>
<tr>
<td>sailboat</td>
<td>motorboat</td>
<td>true or false</td>
<td>no</td>
</tr>
<tr>
<td>unknown</td>
<td>motorboat</td>
<td>false</td>
<td>no</td>
</tr>
<tr>
<td>unknown</td>
<td>motorboat</td>
<td>true</td>
<td>yes</td>
</tr>
</tbody>
</table>

The strict_ignore parameter, as with other filtering parameters, is also used in configuring the contact manager, applied globally and overriding any local filter behavior configurations.

### 23.2.4 Failing an Exclusion Filter on Spawned Behavior

The primary use of exclusion filters is to affect which behaviors are spawned. However, the configured criteria is based on the vehicle properties of group, name, platform type and location, and these properties may change after a behavior is spawned. Normally a contact behavior does not complete, and exit, until it has gone out of range. The user may also opt to have the behavior exit if its properties change after spawning such that it no longer satisfies the original exclusion filter.

A contact behavior will accept the following three configuration parameters:

- **exit_on_filter_group**: If true, will apply the current known group name to an exclusion filter.
- **exit_on_filter_vtype**: If true, will apply the current known vehicle to an exclusion filter.
- **exit_on_filter_region**: If true, will apply the current known vehicle position to the region component of an exclusion filter.

The default for all parameters is false. A common example is when a detected change of group name or vehicle type is learned via communications or sensors. A vehicle may also move into a region where a behavior may be configured to be no longer concerned with a contact. Failing the exclusion filter in a spawned behavior will simply result in the exit/removal of the behavior from the helm, and deletion of the behavior from memory space.

Unlike the group, vehicle type and region components of an exclusion filter, the vehicle name component of the exclusion filter is not re-assessed after a behavior has spawned, since contact behaviors are essentially keyed by vehicle name.

While the above three parameters are supported by all contact behaviors, not all contact behaviors respect this additional feature. Currently (Aug 2021) only the collision avoidance behaviors (CPA based and COLREGS based) support this feature. Support in additional behaviors will be rolled in on future updates.
23.3 Contact Flags

Available after Release 19.8.1, contact flags are an additional way to configure behaviors to post configurable information upon events based on the relative position between ownship and the contact. Most other types of behavior flags contain simply a variable value pair. For example:

```plaintext
endflag = RETURN = true
```

Contact flags allow the user to check for expected results over the course of a mission. It also allows missions to be configured on contact related events such as posting a message to command-and-control or to another vessel when the relative position or range to a contact has been achieved.

23.3.1 Contact Flag Trigger Tags

With contact flags, a trigger tag is included with the flag configuration corresponding to certain events. For example, the @cpa trigger tag will ensure the configured flag will be posted at the time of an observed CPA between ownship and the contact:

```plaintext
cnflag = @cpa SEND_MESSAGE = true
```

The trigger tag is always at the beginning of the flag configuration with at least one white space between the tag and the rest of the flag configuration. The trigger tags only apply to contact flags, for contact behaviors, using the cnflag parameter. Supported trigger flags:

<table>
<thead>
<tr>
<th>Trigger Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>@cpa</td>
<td>When the closes point of approach is observed</td>
</tr>
<tr>
<td>@os_passes_cn</td>
<td>When ownship passes contact’s beam</td>
</tr>
<tr>
<td>@os_passes_cn_port</td>
<td>When ownship passes contact’s port beam</td>
</tr>
<tr>
<td>@os_passes_cn_star</td>
<td>When ownship passes contact’s starboard beam</td>
</tr>
<tr>
<td>@cn_passes_os</td>
<td>When contact passes ownship’s beam</td>
</tr>
<tr>
<td>@cn_passes_os_port</td>
<td>When contact passes ownship’s port beam</td>
</tr>
<tr>
<td>@cn_passes_os_star</td>
<td>When contact passes ownship’s starboard beam</td>
</tr>
<tr>
<td>@os_crosses_cn</td>
<td>When ownship crosses ownship’s side</td>
</tr>
<tr>
<td>@os_crosses_cn_bow</td>
<td>When ownship crosses ownship’s side fore of ownship</td>
</tr>
<tr>
<td>@os_crosses_cn_stern</td>
<td>When ownship crosses ownship’s side aft of ownship</td>
</tr>
<tr>
<td>@cn_crosses_os</td>
<td>When contact crosses ownship’s side</td>
</tr>
<tr>
<td>@cn_crosses_os_bow</td>
<td>When contact crosses ownship’s side fore of ownship</td>
</tr>
<tr>
<td>@cn_crosses_os_stern</td>
<td>When contact crosses ownship’s side aft of ownship</td>
</tr>
</tbody>
</table>

An example is given in Figure 48 below of an encounter between ownship and a contact. The relevant trigger tags are shown.
Figure 48: **An example encounter with trigger tags:** As a contact passes ownship various trigger tags become satisfied. The first two trigger tags are based purely on range and are triggered as the vehicle enters the range. The last trigger tag is triggered as the contact opens range to 60. As the two vehicles pass, several trigger tags are satisfied simultaneously.

### 23.3.2 Contact Flag Macros

Contact flags have a number of supported macros that may be expanded in any value component of a posting. For example, the range between ownship and the contact can be used in a `cnflag` posting:

```plaintext
 cnflag = @cpa RANGE_TO_CONTACT = $[RANGE]
```

The value of $[RANGE]$ is determined at the moment the flag is triggered. Supported macros:

<table>
<thead>
<tr>
<th>MACROS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[RANGE]$</td>
<td>Range between ownship and contact</td>
</tr>
<tr>
<td>$[CN_NAME]$</td>
<td>Name of the contact</td>
</tr>
<tr>
<td>$[CN_GROUP]$</td>
<td>Name of the contact</td>
</tr>
<tr>
<td>$[CN_VTYPE]$</td>
<td>Vehicle type of the contact</td>
</tr>
<tr>
<td>$[ROC]$</td>
<td>Rate of Closure between ownship and the contact</td>
</tr>
<tr>
<td>$[OS_CN_REL_BNG]$</td>
<td>Relative bearing of the contact to ownship</td>
</tr>
<tr>
<td>$[CN_OS_REL_BNG]$</td>
<td>Relative bearing of ownship to the contact</td>
</tr>
<tr>
<td>$[BNG_RATE]$</td>
<td>Bearing Rate</td>
</tr>
<tr>
<td>$[CN_SPD_IN_OS_POS]$</td>
<td>Speed of contact in the direction of ownship position</td>
</tr>
<tr>
<td>$[OS_FORE_OF_CN]$</td>
<td>true if ownship is currently fore of the contact</td>
</tr>
<tr>
<td>$[OS_AFT_OF_CN]$</td>
<td>true if ownship is currently aft of the contact</td>
</tr>
<tr>
<td>$[OS_PORT_OF_CN]$</td>
<td>true if ownship is currently on port side of the contact</td>
</tr>
<tr>
<td>$[OS_STAR_OF_CN]$</td>
<td>true if ownship is currently on starboard side of the contact</td>
</tr>
<tr>
<td>$[CN_FORE_OF_OS]$</td>
<td>true if the contact is currently fore of ownship</td>
</tr>
<tr>
<td>$[CN_AFT_OF_OS]$</td>
<td>true if the contact is currently aft of ownship</td>
</tr>
</tbody>
</table>
23.4 Properties Common to All Contact Related Behaviors

Contact related behaviors are distinct from non contact related behaviors in that they share a fair amount of functionality in dealing with their contact of interest. Contact related behaviors are implemented as a subclass of the IvPContactBehavior class, which is a subclass of the IvPBehavior class. Much of the shared functionality of contact related behaviors is implemented in the former. The shared functionality includes several common configuration parameters, and mechanisms for reasoning about the closest point of approach (CPA) between the platform and contact for candidate maneuver considerations. These topics are discussed next, prior to the sections on the behaviors themselves.

23.4.1 Common Behavior Configuration Parameters

The following set of parameters are common to all the contact related behaviors:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bearing_lines</td>
<td>If true, a visual artifact will be produced for rendering the bearing line</td>
</tr>
<tr>
<td></td>
<td>between ownership and the contact when the behavior is running. Not all</td>
</tr>
<tr>
<td></td>
<td>behaviors implement this feature.</td>
</tr>
<tr>
<td>contact</td>
<td>Name or unique identifier of a contact to be avoided.</td>
</tr>
<tr>
<td>decay</td>
<td>Time interval during which extrapolated position slows to a halt.</td>
</tr>
<tr>
<td>exit_on_filter_group</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior,</td>
</tr>
<tr>
<td></td>
<td>an early exit of the behavior may be allowed when or if the group name</td>
</tr>
<tr>
<td></td>
<td>changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_vtype</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior,</td>
</tr>
<tr>
<td></td>
<td>an early exit of the behavior may be allowed when or if the vehicle type</td>
</tr>
<tr>
<td></td>
<td>changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_region</td>
<td>It true, and if an exclusion filter is implemented for this contact behavior,</td>
</tr>
<tr>
<td></td>
<td>an early exit of the behavior may be allowed when or if the contact moves</td>
</tr>
<tr>
<td></td>
<td>into a region that would no longer satisfy the exclusion filter. The default</td>
</tr>
<tr>
<td></td>
<td>it false.</td>
</tr>
<tr>
<td>extrapolate</td>
<td>If true, contact position is extrapolated from last position and trajectory.</td>
</tr>
<tr>
<td>ignore_group</td>
<td>If specified, the contact group may not match the given ignore group. If</td>
</tr>
<tr>
<td></td>
<td>multiple ignore groups are specified, the contact group must be different</td>
</tr>
<tr>
<td></td>
<td>than all ignore groups. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>ignore_name</td>
<td>If specified, the contact name may not match the given ignore name. If</td>
</tr>
<tr>
<td></td>
<td>multiple ignore names are specified, the contact name must be different</td>
</tr>
<tr>
<td></td>
<td>than all ignore names. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
</tbody>
</table>
The **contact** parameter specifies the contact name or identifier. This name is used as a key by the behaviors for querying the contact position and trajectory. It must match the contact name received by the helm in an incoming `NODE_REPORT` message. The contact name may be specified at helm launch time, but it may also be specified at run time if the behavior is configured as a template for dynamic spawning. The latter is more common, for example, in a collision avoidance behavior where the name or ID of the contact is not known until a contact manager alerts the helm. See the Berta mission in Section 34 for an example of this usage.

The **extrapolate** and **decay** parameters are used to address the situation where a contact/node report has significant delays between updates. Extrapolation is enabled by setting the `extrapolate` parameter to `true`, which is the default. The behavior may be configured to have limited extrapolation by setting a decay time interval. The extrapolated position is based on the last known contact position, heading and speed. The speed used for calculations may begin decaying at the beginning of the decay interval and will have decayed to zero at the end of the decay interval. The default setting is "decay = 15, 30", in seconds. The idea is shown in Figure 49.
Figure 49: Contact Extrapolations: Contact related behaviors may use an extrapolated position of the contact to compensate for periods of no new reports for the contact. A decay period may be used to effectively halt the extrapolated contact position after some specified period of time. In the example in this figure, the decay window is [15, 30] seconds. After 30 seconds, the extrapolated position does not extend further.

The on_no_contact_ok parameter determines how the behavior should regard the situation where it is unable to find any information about a given contact. If this parameter is set to true, the default, then the behavior will post a warning, BHV_WARNING if no contact information is found. Otherwise the behavior will post an error with BHV_ERROR. In the latter case the helm may interpret this as request to halt the helm and come to zero speed and depth.

The time_on_leg parameter refers to the behavior’s calculations of the closest point of approach (CPA) for candidate maneuver legs. A candidate maneuver leg is defined by a the heading, speed, and time-on-leg components. Longer time-on-leg settings tend to report deceivingly worrisome CPA distances even for contacts at a great distance, and lower time-on-leg settings tend to report deceivingly comfortable CPA distances even for vehicles at relatively low distances. The default setting for this parameter is 60 seconds.

23.4.2 Closest Point of Approach Calculations

The IvP functions produced by contact-related behaviors are defined over the domain of possible heading and speed choices. The utility assigned to a point in this domain (a heading-speed pair) depends in part on the calculated closest point of approach (CPA) between the candidate maneuver leg, and the contact leg formed from the contact’s position and trajectory. Figure 50 shows the relationship cpa(θ, v) between CPA and candidate maneuvers (θ, v), where θ=heading and v=speed, for a given relative position between ownship and a given contact vehicle and trajectory. The IvP function generated by the AvoidCollision behavior applies a further user-defined utility function to the CPA calculation for a candidate maneuver, f(cpa(θ, v)). The form of f() is determined by configuration parameters specific to the individual behavior.
For contact related behaviors, an important quality of a candidate action \((\theta, v, t)\), is the closest point of approach (CPA) between two vehicles during a candidate leg. A behavior producing an objective function with CPA as a component of its utility function needs to perform many variations of this calculation on each new call to generate an IvP objective function. The algorithm is given here, highlighting a few areas where caching may be exploited to improve efficiency.

Our own current position is known and given by \((x, y)\), and the other vehicle’s current position and trajectory is given by \((x_b, y_b, \theta_b, v_b)\). To compute the CPA distance for a given \((\theta, v, t)\), first the time \(t_{\text{min}}\) when the minimum distance between two vehicles occurs is computed. The distance between the two vehicles at the current time can be determined by the Pythagorean theorem. Generally, for any given time \(t\) (where the current time is \(t = 0\)), and assuming the other vehicle stays on a constant trajectory, the distance between the two vehicles for any chosen \((\theta, v, t)\) is given by:

\[
\text{dist}^2(\theta, v, t) = k_2 t^2 + k_1 t + k_0,
\]

where

\[
k_2 = \cos^2(\theta) v^2 - 2 \cos(\theta) v \cos(\theta_b) v_b + \cos^2(\theta_b) v_b^2 + \sin^2(\theta) v^2 - 2 \sin(\theta) v \sin(\theta_b) v_b + \sin^2(\theta_b) v_b^2
\]

\[
k_1 = 2 \cos(\theta) v y - 2 \cos(\theta) v y_b - 2 y \cos(\theta_b) v_b + 2 \cos(\theta_b) v_b y_b + 2 \sin(\theta) v x - 2 \sin(\theta) v x_b - 2 x \sin(\theta_b) v_b + 2 \sin(\theta_b) v_b x_b
\]

\[
k_0 = y^2 - 2 y y_b + y_b^2 + x^2 - 2 x x_b + x_b^2
\]

The stationary point is obtained by taking the first derivative with respect to \(t\):
\[ \text{dist}^2(\theta, v, t)' = 2k_2 t + k_1. \]

Since there is no maximum distance, this stationary point always represents the closest point of approach, and therefore:

\[ t' = \frac{-k_1}{2k_2}. \]

The value of \( t_{min} \) may be in the past, i.e., less than zero, if the two vehicles are currently opening range. Or \( t_{min} \) may be well beyond \( t \), the time length of the candidate maneuver \( \langle \theta, v, t \rangle \). Therefore the value of \( t_{min} \) is clipped by \([0, t] \). Furthermore \( t_{min} \) is zero when the two vehicles have the same heading and speed (the only condition where \( k_2 \) is zero). The actual CPA value is then obtained by plugging \( t_{min} \) back into first equation above.

\[ \text{CPA}(\theta, v, t) = \sqrt{k_2 t_{min}^2 + k_1 t_{min} + k_0}. \]

As mentioned before, this calculation is a common component in the underlying utility function for behaviors dealing with relative vehicle motion. A behavior, within a single iteration of the control cycle, will perform a sequence of calculations on different \( \langle \theta, v, t \rangle \) values. However, all calculations have the same values of current vehicle position \((x, y)\), and current position and trajectory of the other vehicle \((x_b, y_b, \theta_b, v_b)\). To make this overall sequence of calculations faster, all terms in \( k_0, k_1, \) and \( k_2 \) above comprised exclusively of \( x, y, x_b, y_b, \theta_b, v_b \) are calculated once and cached for later calculations.
24  The AvoidCollision Behavior

The AvoidCollision behavior will produce IvP objective functions designed to avoid collisions (and near collisions) with another specified vehicle. The IvP functions produced by this behavior are defined over the domain of possible heading and speed choices. The utility assigned to a point in this domain (a heading-speed pair) depends in part on the calculated closest point of approach (CPA) between the candidate maneuver leg, and the contact leg formed from the contact’s position and trajectory. A further user-defined utility function is applied to the CPA calculation for a candidate maneuver.

24.1  Configuration Parameters

Listing 24.1: Configuration Parameters Common to All Behaviors.

- `activeflag`: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- `condition`: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- `duration`: Time in behavior will remain running before declaring completion. Section 7.2.6.
- `duration_idle_decay`: When true, duration clock is running even when in the idle state. Section 7.2.6.
- `duration_reset`: A variable-pair such as `MY_RESET=true`, that will trigger a duration reset. See Section 7.2.6.
- `duration_status`: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- `endflag`: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- `idleflag`: A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
- `inactiveflag`: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
- `name`: The (unique) name of the behavior. Section 7.2.2.
- `nostarve`: Allows a behavior to assert a maximum staleness for a MOOS variable. Section 7.2.9.
- `perpetual`: If true allows the behavior to to run even after it has completed. Section 7.2.7.
- `post_mapping`: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- `priority`: The priority weight of the behavior. Section 7.2.3.
- `put`: Same as `priority`.
- `runflag`: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 24.2: Configuration Parameters Common to Contact Behaviors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bearing_lines</td>
<td>If true, a visual artifact will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.</td>
</tr>
<tr>
<td>contact</td>
<td>Name or unique identifier of a contact to be avoided.</td>
</tr>
<tr>
<td>decay</td>
<td>Time interval during which extrapolated position slows to a halt.</td>
</tr>
<tr>
<td>exit_on_filter_group</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_vtype</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_region</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>extrapolate</td>
<td>If true, contact position is extrapolated from last position and trajectory.</td>
</tr>
<tr>
<td>ignore_group</td>
<td>If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>ignore_name</td>
<td>If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>ignore_region</td>
<td>If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>ignore_type</td>
<td>If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>match_group</td>
<td>If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
</tbody>
</table>
match_name: If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 23.2

match_region: If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regions. Introduced after Release 19.8.1. Section 23.2

match_type: If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 23.2

on_no_contact_ok If false, a helm error is posted if no contact information exists. Applicable in the more rare case that a contact behavior is statically configured for a named contact. The default is true.

strict_ignore If true, and if one of the ignore exclusion filter components is enabled, then an exclusion filter will fail if the contact report is missing information related to the filter. For example if the contact group information is unknown. The default is true.

time_on_leg The time on leg, in seconds, used for calculating closest point of approach.

Listing 24.3: Configuration Parameters for the AvoidCollision Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>completed_dist</td>
<td>Range to contact outside of which the behavior completes and dies. The default is 500 meters.</td>
</tr>
<tr>
<td>max Util CPA dist</td>
<td>Range to contact outside which a considered maneuver will have max utility. Section 24.6. The default is 75 meters</td>
</tr>
<tr>
<td>min Util CPA dist</td>
<td>Range to contact within which a considered maneuver will have min utility. Section 24.6. The default is 10 meters.</td>
</tr>
<tr>
<td>no alert request</td>
<td>If true, the behavior will not send an automatic configuration message to the contact manager at startup. Section 24.4. The default is false.</td>
</tr>
<tr>
<td>pwt_grade</td>
<td>Grade of priority growth as the contact moves from the pwt_outer_dist to the pwt_inner_dist. Choices are linear, quadratic, or quasi. The default is quasi. Section 24.5.</td>
</tr>
<tr>
<td>pwt_inner_dist</td>
<td>Range to contact within which the behavior has maximum priority weight. Section 24.5. The default is 50 meters.</td>
</tr>
<tr>
<td>pwt_outer_dist</td>
<td>Range to contact outside which the behavior has zero priority weight. Section 24.5. The default is 200 meters.</td>
</tr>
<tr>
<td>use_refinery</td>
<td>If true, the behavior will produce an optimized objective function that is faster to produce, uses a smaller memory footprint, and contributes to faster helm solution time. The default is false, simply for continuity with prior releases, but there is no downside to enabling this feature. Section 24.8.</td>
</tr>
</tbody>
</table>

Listing 24.4: Example Configuration Block.
Behavior = BHV_AvoidCollision
{
    // General Behavior Parameters
    // ---------------------------
    name = avdcollision_ // example
    pwt = 200 // example
    condition = AVOID = true // example
    updates = CONTACT_INFO // example
    endflag = CONTACT_RESOLVED = [$CONTACT] // example
    templating = spawn // example

    // General Contact Behavior Parameters
    // -----------------------------------
    bearing_lines = white:0, green:0.65, yellow:0.8, red:1.0 // example
    contact = henry // example
    decay = 15,30 // default (seconds)
    extrapolate = true // default
    on_no_contact_ok = true // default
    time_on_leg = 60 // default (seconds)

    // Parameters specific to this behavior
    // ------------------------------------
    completed_dist = 500 // default
    max_util_cpa_dist = 75 // default
    min_util_cpa_dist = 10 // default
    no_alert_request = false // default
    pwt_grade = quasi // default
    pwt_inner_dist = 50 // default
    pwt_outer_dist = 200 // default
}

24.2 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be suppressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.8.

- BCM_ALERT_REQUEST: A request to the contact manager specifying the conditions for contact alerts.
- CLOSING_SPD_AVD: The current closing speed, in meters per second, to the contact.
- CONTACT_RESOLVED: Posted with contact name when the behavior completes and dies.
- RANGE_AVD: The current range, in meters, to the contact.
- VIEW_SEGLIST: A bearing line between ownship and the contact if configured for rendering.

24.3 Configuring and Using the AvoidCollision Behavior

The AvoidCollision behavior produces an objective function based on the relative positions and trajectories between the vehicle and a given contact. The objective function is based on applying a
utility to the calculated closest point of approach (CPA) for a candidate maneuver. The user may configure a priority weight, but this weight is typically degraded as a function of the range to the contact. The behavior may be configured for avoidance with respect to a known contact, or it may be configured to spawn a new instance upon demand as contacts present themselves.

### 24.4 Automatic Requests for Contact Manager Alerts

The collision avoidance behavior is most commonly configured as a *template*, meaning instances will not be spawned until an outside event, i.e., posting to the MOOSDB, is received by the helm. This was discussed in detail in Section 7.7. The spawning event for the collision avoidance behavior typically comes from the `pBasicContactMgr` application. This app therefore needs to be informed, by the collision avoidance behavior, of the desired conditions for generating behavior-spawning alerts. This is done automatically upon helm startup with a posting of the form:

```
BCM_ALERT_REQUEST = id=avd, onflag=CONTACT_INFO=name=$[VNAME] # contact=$[VNAME], alert_range=80, cpa_range=100
```

The values for this posting are chosen as follows:

- The value from the `onflag` component is a posting to be made by the contact manager when an alert is triggered. Like all MOOS postings, the posting has a variable and value component. The variable component is `CONTACT_INFO`. The value component is `name=$[VNAME] # contact=$[VNAME]`

- When the contact manager posts the onflag, it will expand the `$[VNAME]` macros with the actual name of the contact. The variable `CONTACT_INFO` is the variable provided in the `updates` parameter for the behavior.

- The value from the `alert_range` component, e.g., 80 in the example above, is the value specified in the `pwt_outer_dist` parameter for the behavior. This is the range between ownship and contact beyond which the behavior assigns a priority weight of zero.

- The value from the `cpa_range` component, e.g., 100 in the example above, is the value specified in the `completed_dist` parameter for the behavior. This is the range, in meters, between ownship and contact beyond which the behavior will initiate its own completion and de-instatiation.

If some other contact manager regime is being used other than `pBasicContactMgr`, the above automatic posting is probably harmless. However, if one really wants to disable this automatic posting, it can be turned off by setting the configuration parameter `no_alert_request` to true.

**Implementation note:** One may wonder when or how the behavior can make this automatic posting when the behavior is configured as a template. An instance is never spawned until an alert is received, but the alert parameters are posted by the behavior, creating a bit of a chicken or the egg conundrum. The alert request is actually posted by an instance of the behavior created ever so briefly at helm startup. At startup, the helm creates instances of _all_ behaviors, even templates, to ensure the configuration parameters are correct. The ultimate confirmation of behavior parameter correctness is obtained by a behavior instance itself confirming each parameter. The helm will then immediately, before its first iteration, delete any behaviors temporarily created from template
behaviors. During this brief startup period, the helm will invoke the function `onHelmStart()` for all behaviors. This function is defined at the IvPBehavior superclass level just like `onRunState()`. In the case of the collision avoidance behavior, this function is implemented to make the automatic alert configuration posting to `BCM_ALERT_REQUEST`.

### 24.5 Specifying the Behavior Priority Weight Policy

The AvoidCollision behavior may be configured to increase its priority as it closes range to the contact. The priority weight specified in its configuration represents the maximum possible priority applied to the behavior, presumably in close range to the contact. The range at which this maximum priority applies is specified in the `pwt_inner_dist` parameter. Likewise, the `pwt_outer_dist` parameter specifies a range to the contact where the priority weight becomes zero, regardless of the priority weight specified in the configuration file.

So the current priority will always be between zero and the maximum priority set in the behavior priority configuration parameter. To be more precise:

Current Priority =

- 0 if current range to contact is greater than or equal to `pwt_outer_dist`
- 100 if current range to contact is less than or equal to `pwt_inner_dist`
- otherwise `((pwt_outer_dist - current range) / (pwt_outer_dist - pwt_inner_dist)) * priority`

This relationship is shown in Figure 51.

![Figure 51: Scaling priority weights based on ownership range to contact](image)

Figure 51: Scaling priority weights based on ownership range to contact: The range between the two vehicles affects whether the behavior is spawned, active and with what priority weight. Beyond the range specified by `completed_dist` the behavior is likely not inexistence. Beyond the range specified by `pwt_outer_dist`, the behavior is not yet active. Within the range of `pwt_outer_dist`, the behavior becomes active with a non-zero priority weight growing as the contact closes range. Within the range of `pwt_inner_dist`, the behavior is active with 100% of its configured priority weight.
The example shown below in Figure 52 shows the effect of the \texttt{pwt\_outer\_dist} parameter. The vehicle on the left is proceeding east, oblivious to the two approaching vessels. The two westbound vessels, \texttt{ben} and \texttt{cal} are simulated exactly on top of one another. They are oblivious to one another, but will use the collision avoidance behavior to avoid the eastbound vessel, \texttt{abe}. The only difference between \texttt{ben} and \texttt{cal} is that \texttt{cal} begins winding up its priority weight at 80 meters range to \texttt{abe}, as opposed to 30 meters for \texttt{ben}. The short simulation shows the resulting difference in trajectory.

![Figure 52: Two vehicles use the AvoidCollision behavior to avoid the oblivious and non-maneuvering Eastbound vehicle. One vehicle has \texttt{pwt\_outer\_dist} set to 80 and the other set to 30. This affects when each vehicle begins its maneuver to avoid.](image)

By default, the priority weight decreases linearly between the two depicted ranges. The \texttt{pwt\_grade} parameter allows the degradation from maximum priority to zero priority to fall more steeply by setting \texttt{pwt\_grade=quadratic}.

### 24.6 Specifying the Utility Policy of the Behavior

Whereas the behavior \textit{weight} discussed above in Section 24.5 determines the influence of the behavior relative to other behaviors, the behavior \textit{utility function} specifies the relative utility of candidate maneuvers from the perspective of the collision avoidance goals of this behavior.

The utility function for the avoid collision behavior is based on two factors:

- The range at the closest point of approach (CPA) for a candidate maneuver
• The determination risk associated with any CPA range.

The first component is just physics. Given ownship and contact current positions and trajectories, and the assumption that the contact will stay on its current heading and speed for at least the near future, the projected CPA range may be calculated for any given heading-speed maneuver. Consider the example in Figure 53 below with particular ownship and contact positions and trajectories shown on the left. On the right, the projected CPA range values are plotted for all candidate ownship maneuvers.

The second component of the utility function is the mapping of "risk" to certain CPA range values. This part is very subjective, and of course is why there are configuration parameters, allow different users to align the behavior with their own notion of risk. The two key parameters are \( \text{min}_\text{util}_{\text{cpa\_dist}} \) and \( \text{max}_\text{util}_{\text{cpa\_dist}} \). The former refers the the CPA range with the minimum utility, essentially equivalent to a collision. For this value, perhaps pick a range that, while not a collision, someone would get written up for a safety violation for breaching this range. The \( \text{max}_\text{util}_{\text{cpa\_dist}} \) range, on the other hand, is the range that, above which there is not increase in utility. Everything at this range or higher is "all clear".

These two parameters form a function, \( g() \), which takes as input the CPA range value and outputs the utility of a candidate maneuver. This function is depicted on the left in Figure 54, and the overall utility function is shown on the right:
Figure 54: (left) A utility function mapping CPA range values to a single risk utility is shown. Up to the range specified by the \text{min\_util\_cpa} configuration parameter, maneuvers resulting in these ranges are considered essentially collisions. Above the range specified by the \text{max\_util\_cpa} configuration parameter, maneuvers resulting in these ranges are considered essentially equivalently optimal. In between are the ranges that are in between disaster and optimal. These represent compromise maneuvers that the helm may opt for if there are other behaviors that find such maneuvers useful for accomplishing their objectives.

The \text{min\_util\_cpa\_dist} and \text{max\_util\_cpa\_dist} parameters have a default value of 10 and 75 meters respectively. These values are very subjective and will need to be adjusted per vehicle and mission. Currently these defaults are used if not specified by the user, but in future releases overriding these values may be enforced.
Figure 55: Two vehicles use the AvoidCollision behavior to avoid the oblivious and non-maneuvering eastbound vehicle. One vehicle, ben, is configured with the \texttt{min\_util\_cpa\_dist} and \texttt{max\_util\_cpa\_dist} parameters set to 5 and 15 meters respectively. The other vehicle, cal, is more risk averse and has these to parameters set to 15 and 25 meters respectively. The resulting trajectories of ben and cal are shown. The two vehicles on the right are simulated separately, unaware of each other, with the same starting position and parameters except for two above parameters. The divergence in trajectory is solely due to the differences around these two parameters.

\texttt{video:\(0:13\):} \url{https://vimeo.com/458207817}

24.7 Specifying Contact Flags

Contact flags are new feature for all contact behaviors, available in releases after Release 19.8.1.

24.8 Using the CPA Refinery

24.9 Relevant Example Missions
25 The CutRange Behavior

The CutRange behavior will drive ownship to reduce the range between itself and another specified vehicle (nearly the opposite of the BHV_AvoidCollision behavior).

25.1 The Patience Parameter

The behavior reasons about candidate ownship maneuvers and will use a combination of immediate-rate-of-closure (IROC) and expected closest point of approach (CPA) distance, to evaluate candidate maneuvers. Using solely IROC is regarded as having zero patience, and using solely CPA is regarded as having maximal patience. The idea is conveyed in Figure 56 below. The former method may be preferred for a contact with frequent heading changes, while the latter may be preferred for contacts on a trajectory that rarely changes.

![Figure 56: The CutRange behavior will either drive ownship directly toward the current contact position (zero patience setting) or drive ownship to be on the fastest intercept course based on the contact current position, heading and speed (maximal patience setting).](image)

25.2 Automatic Priority Weight Variation Based on Range

The CutRange behavior may optionally be configured to automatically decrease its priority weight down to zero as it closes range to the contact. This can be used to ensure ownship does not collide with the contact or to transition the helm to a different mode of operation as it closes range to the contact. The idea is shown in Figure 57.
When ownship is at the outer range \((pwt\_outer\_range)\), the priority is at its maximal value. When ownship is at the inner most range \((pwt\_inner\_range)\), the priority weight drops to zero. The priority weight decreases linearly as the range is closed. If the overall configured priority weight for the behavior were set to 80 instead of 100, the outer weight would be 80, and 40 at the intermediate range.

When the contact is at a range at or beyond \(pwt\_outer\_dist\), the priority weight will be 100 percent of the behavior’s configured priority weight. When the contact is at or closer than the range given by \(pwt\_inner\_dist\) the priority weight will be zero. At contact ranges between these two values the priority weight will linearly decrease in value. In the situation depicted in Figure 57, the vehicle transition to a dominant Shadow behavior as the range is closed. When the range decreases to \(pwt\_inner\_dist\), the Shadow behavior is solely determining ownship heading and speed to match the contact’s current heading and speed.

### 25.3 Initiating and Abandoning Pursuit

The \(giveup\_dist\) parameter sets a contact range beyond which the behavior will assign, a zero priority weight, essentially giving up pursuit and becoming an inactive behavior. The parameter is typically set to be a bit higher than the \(pwt\_outer\_dist\) parameter, but any non-zero configured value will be respected. A \(giveup\_dist\) of zero is the default, and disables this feature.

As ownship range to the contact increases and crosses the \(giveup\_dist\) threshold, the behavior may be configure to post one or more messages, using the \(giveupflag\) parameter. Similarly, when ownship range to contact decreases below the giveup range, the behavior may be configure to post one or more messages, using the \(pursueflag\) parameter. For example:

```
pursueflag = PURSUIT=true
giveupflag = PURSUIT=false
```

In the above case, the two flags could be used to transition the helm out of its present mission mode and back again based on the status of closing range to the contact.
25.4 Configuration Parameters

Listing 25.1: Configuration Parameters Common to All Behaviors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activeflag</td>
<td>A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.</td>
</tr>
<tr>
<td>condition</td>
<td>Specifies a condition that must be met for the behavior to be running. Section 6.5.1.</td>
</tr>
<tr>
<td>duration</td>
<td>Time in behavior will remain running before declaring completion. Section 7.2.6.</td>
</tr>
<tr>
<td>duration_idle_decay</td>
<td>When true, duration clock is running even when in the idle state. Section 7.2.6.</td>
</tr>
<tr>
<td>duration_reset</td>
<td>A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.</td>
</tr>
<tr>
<td>duration_status</td>
<td>The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.</td>
</tr>
<tr>
<td>endflag</td>
<td>A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.</td>
</tr>
<tr>
<td>idleflag</td>
<td>A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.</td>
</tr>
<tr>
<td>name</td>
<td>The (unique) name of the behavior. Section 7.2.2.</td>
</tr>
<tr>
<td>nostarve</td>
<td>Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.</td>
</tr>
<tr>
<td>perpetual</td>
<td>If true allows the behavior to run even after it has completed. Section 7.2.7.</td>
</tr>
<tr>
<td>post_mapping</td>
<td>Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.</td>
</tr>
<tr>
<td>priority</td>
<td>The priority weight of the behavior. Section 7.2.3.</td>
</tr>
<tr>
<td>pwt</td>
<td>Same as priority.</td>
</tr>
<tr>
<td>runflag</td>
<td>A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.</td>
</tr>
<tr>
<td>spawnflag</td>
<td>A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.</td>
</tr>
<tr>
<td>templating</td>
<td>Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.</td>
</tr>
<tr>
<td>updates</td>
<td>A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.</td>
</tr>
</tbody>
</table>

Listing 25.2: Configuration Parameters Common to Contact Behaviors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>

200
**bearing_lines** If true, a visual artifact will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.

**contact** Name or unique identifier of a contact to be avoided.

**decay** Time interval during which extrapolated position slows to a halt.

**exit_on_filter_group** If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.

**exit_on_filter_vtype** If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.

**exit_on_filter_region** If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default is false.

**extrapolate** If true, contact position is extrapolated from last position and trajectory.

**ignore_group:** If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 23.2

**ignore_name:** If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 23.2

**ignore_region:** If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 23.2

**ignore_type:** If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 23.2

**match_group:** If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 23.2

**match_name:** If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one.Introduced after Release 19.8.1. Section 23.2

**match_region:** If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regions. Introduced after Release 19.8.1. Section 23.2

**match_type:** If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 23.2
on_no_contact_ok  If false, a helm error is posted if no contact information exists. Applicable
in the more rare case that a contact behavior is statically configured for a
named contact. The default is true.

strict_ignore  If true, and if one of the ignore exclusion filter components is enabled, then
an exclusion filter will fail if the contact report is missing information related
to the filter. For example if the contact group information is unknown.
The default is true.

time_on_leg  The time on leg, in seconds, used for calculating closest point of approach.

Listing 25.3: Configuration Parameters for the CutRange Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>giveup_dist</td>
<td>Range to contact, in meters, outside which the behavior will give up (become inactive). Default is zero. Section 25.3.</td>
</tr>
<tr>
<td>giveupflag</td>
<td>Optional MOOS variable-value pairs posted when ownship range to contact increases above giveup_dist. Section 25.3.</td>
</tr>
<tr>
<td>patience</td>
<td>Linear scale choice between preferring heading directly to the contact (patience=0) or heading on the lowest closest point of approach (patience=100). Section 25.1.</td>
</tr>
<tr>
<td>pursueflag</td>
<td>Optional MOOS variable-value pairs posted when ownship range to contact drops below giveup_dist. Section 25.3.</td>
</tr>
<tr>
<td>pwt_outer_dist</td>
<td>Range to contact outside which the behavior has maximum priority. Section 25.2.</td>
</tr>
<tr>
<td>pwt_inner_dist</td>
<td>Range to contact within which the behavior has zero priority. Section 25.2.</td>
</tr>
</tbody>
</table>

Listing 25.4: Example Configuration Block.
Behavior = BHV_CutRange
{
    // General Behavior Parameters
    // ---------------------------
    name = cutrange_ // example
    pwt = 200 // example
    condition = AVOID = true // example
    updates = CONTACT_INFO // example
    endflag = CONTACT_RESOLVED = $[CONTACT] // example
    templating = spawn // example

    // General Contact Behavior Parameters
    // -----------------------------------
    bearing_lines = white:0, green:0.65, yellow:0.8, red:1.0 // example
    contact = henry // example
    decay = 15,30 // default (seconds)
    extrapolate = true // default
    on_no_contact_ok = true // default
    time_on_leg = 60 // default (seconds)

    // Parameters specific to this behavior
    // ------------------------------------
    giveup_dist = 225 // Meters. Default is 0, disabled
    patience = 45 // [0,100]. Default 0
    pwt_inner_dist = 50 // Meters. Default is 0.
    pwt_outer_dist = 200 // Meters. Default is 0.
    pursueflag = PURSUIT=true
giveupflag = PURSUIT=false
}

25.5 Variables Published
The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be suppressed or changed to a different variable name using the post_mapping configuration parameter described in Section 7.2.8.

- **VIEWSEGLIST**: A bearing line between ownship and the contact if configured for rendering.
- Any variables involved in the giveupflag or pursueflag configurations, or other general behavior flags.
26 The Shadow Behavior

This behavior will drive the vehicle to match the trajectory of another specified vehicle. This behavior in conjunction with the BHV_CutRange behavior can produce a "track and trail" capability.

26.1 Configuration Parameters

Listing 26.1: Configuration Parameters Common to All Behaviors.

- activeflag: A MOOS variable-value pair posted when the behavior is in the active state. Section 6.5.4.
- condition: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- duration: Time in behavior will remain running before declaring completion. Section 7.2.6.
- duration_idle_decay: When true, duration clock is running even when in the idle state. Section 7.2.6.
- duration_reset: A variable-pair such as MY_RESET=true, that will trigger a duration reset. See Section 7.2.6.
- duration_status: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- endflag: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- idleflag: A MOOS variable-value pair posted when the behavior is in the idle state. Section 6.5.4.
- inactiveflag: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.
- name: The (unique) name of the behavior. Section 7.2.2.
- nostarve: Allows a behavior to assert a maximum staleness for a MOOS variable Section 7.2.9.
- perpetual: If true allows the behavior to to run even after it has completed. Section 7.2.7.
- post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.
- priority: The priority weight of the behavior. Section 7.2.3.
- pwt: Same as priority.
- runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.
- spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.
- templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.
**updates**: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

**Listing 26.2: Configuration Parameters Common to Contact Behaviors.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bearing_lines</td>
<td>If true, a visual artifact will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.</td>
</tr>
<tr>
<td>contact</td>
<td>Name or unique identifier of a contact to be avoided.</td>
</tr>
<tr>
<td>decay</td>
<td>Time interval during which extrapolated position slows to a halt.</td>
</tr>
<tr>
<td>exit_on_filter_group</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_vtype</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_region</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default it false.</td>
</tr>
<tr>
<td>extrapolate</td>
<td>If true, contact position is extrapolated from last position and trajectory.</td>
</tr>
<tr>
<td>ignore_group</td>
<td>If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>ignore_name</td>
<td>If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>ignore_region</td>
<td>If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>ignore_type</td>
<td>If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>match_group</td>
<td>If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
<tr>
<td>match_name</td>
<td>If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
</tbody>
</table>
match_region: If specified, the contact must reside in the given convex region. If multiple
match regions are specified, the contact position must be in at least one
match region. The multiple regions essentially can together support a
non-convex regions. Introduced after Release 19.8.1. Section 23.2

match_type: If specified, the contact type must match the given match type. If multiple
match types are specified, the contact type must match at least one match
type. Introduced after Release 19.8.1. Section 23.2

on_no_contact_ok If false, a helm error is posted if no contact information exists. Applicable
in the more rare case that a contact behavior is statically configured for a
named contact. The default is true.

strict_ignore If true, and if one of the ignore exclusion filter components is enabled, then
an exclusion filter will fail if the contact report is missing information related
to the filter. For example if the contact group information is unknown.
The default is true.

time_on_leg The time on leg, in seconds, used for calculating closest point of approach.

Listing 26.3: Configuration Parameters for the Shadow Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>heading_peakwidth</td>
<td>Width of the peak, in degrees, in the produced ZAIC-style IvP function.</td>
</tr>
<tr>
<td>heading_basewidth</td>
<td>Width of the base, in degrees, in the produced ZAIC-style IvP function.</td>
</tr>
<tr>
<td>speed_peakwidth</td>
<td>Width of the peak, in m/sec, in the produced ZAIC-style IvP function.</td>
</tr>
<tr>
<td>speed_basewidth</td>
<td>Width of the base, in m/sec, in the produced ZAIC-style IvP function.</td>
</tr>
</tbody>
</table>

Listing 26.4: Example Configuration Block.
Behavior = BHV_CutRange
{
  // General Behavior Parameters
  // ---------------------------
  name = shadow_henry // example
  pwt = 200 // example
  condition = SHADOW = true // example
  updates = CONTACT_INFO // example

  // General Contact Behavior Parameters
  // -----------------------------------
  contact = henry // example
  decay = 15,30 // default (seconds)
  extrapolate = true // default
  on_no_contact_ok = true // default
  time_on_leg = 60 // default (seconds)

  // Parameters specific to this behavior
  // ------------------------------------
  pwt_outer_dist = 0 // default (meters)
  heading_peakwidth = 20 // default
  heading_basewidth = 160 // default
  speed_peakwidth = 0.1 // default
  speed_basewidth = 2.0 // default
}

- **pwt_outer_dist**: The distance (in meters) that the contact must be within for the behavior to be active and produce an objective function. The default is zero meaning it will be active regardless of the distance to the contact.

- **heading_peakwidth**: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the peakwidth parameter of the heading component.

- **heading_basewidth**: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the basewidth parameter of the heading component.

- **speed_peakwidth**: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the peakwidth parameter of the speed component.

- **speed_basewidth**: This behavior uses the ZAIC_PEAK tool from the IvP Toolbox for generating an objective function over heading and speed. This parameter sets the basewidth parameter of the speed component.

### 26.2 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be suppressed or changed to a
different variable name using the `post_mapping` configuration parameter described in Section 7.2.8.

**Listing 26.5: Variables Published by the Shadow Behavior.**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHADOW_CONTACT_X:</td>
<td>description</td>
</tr>
<tr>
<td>SHADOW_CONTACT_Y:</td>
<td>description</td>
</tr>
<tr>
<td>SHADOW_CONTACT_SPEED:</td>
<td>description</td>
</tr>
<tr>
<td>SHADOW_CONTACT_HEADING:</td>
<td>description</td>
</tr>
<tr>
<td>SHADOW_CONTACT_RELEVANCE:</td>
<td>description</td>
</tr>
</tbody>
</table>
27 The Trail Behavior

This behavior will drive the vehicle to trail or follow another specified vehicle at a given relative position. A tool for "formation flying". The following parameters are defined for this behavior:

![Figure 58: Interpolation of vehicle speed inside the radius set by \textit{nm\_radius} relative to the extrapolated trail position.](image)

27.1 Configuration Parameters

\textit{Listing 27.1: Configuration Parameters Common to All Behaviors.}

- \textit{activeflag}: A MOOS variable-value pair posted when the behavior is in the \textit{active} state. Section 6.5.4.
- \textit{condition}: Specifies a condition that must be met for the behavior to be running. Section 6.5.1.
- \textit{duration}: Time in behavior will remain running before declaring completion. Section 7.2.6.
- \textit{duration\_idle\_decay}: When true, duration clock is running even when in the \textit{idle} state. Section 7.2.6.
- \textit{duration\_reset}: A variable-pair such as \texttt{MY\_RESET=true}, that will trigger a duration reset. See Section 7.2.6.
- \textit{duration\_status}: The name of a MOOS variable to which the vehicle duration status is published. Section 7.2.6.
- \textit{endflag}: A MOOS variable-value pair posted when the behavior has completed. Section 6.5.4.
- \textit{idleflag}: A MOOS variable-value pair posted when the behavior is in the \textit{idle} state. Section 6.5.4.
inactiveflag: A MOOS variable-value posted when the behavior is not in the active state. Section 6.5.4.

name: The (unique) name of the behavior. Section 7.2.2.

nostarve: Allows a behavior to assert a maximum staleness for a MOOS variable. Section 7.2.9.

perpetual: If true allows the behavior to to run even after it has completed. Section 7.2.7.

post_mapping: Re-direct behavior output normally to one MOOS variable to another instead. Section 7.2.4.

priority: The priority weight of the behavior. Section 7.2.3.

pwt: Same as priority.

runflag: A MOOS variable and a value posted when a behavior has met its conditions. Section 6.5.4.

spawnflag: A MOOS variable and a value posted when a behavior is spawned. Section 6.5.4.

templating: Turns a behavior into a template for spawning behaviors dynamically. Section 7.7.

updates: A MOOS variable from which behavior parameter updates are read dynamically. Section 7.2.5.

Listing 27.2: Configuration Parameters Common to Contact Behaviors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bearing_lines</td>
<td>If true, a visual artifact will be produced for rendering the bearing line between ownship and the contact when the behavior is running. Not all behaviors implement this feature.</td>
</tr>
<tr>
<td>contact</td>
<td>Name or unique identifier of a contact to be avoided.</td>
</tr>
<tr>
<td>decay</td>
<td>Time interval during which extrapolated position slows to a halt.</td>
</tr>
<tr>
<td>exit_on_filter_group</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the group name changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_vtype</td>
<td>If true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the vehicle type changes and no longer satisfies the exclusion filter. The default is false.</td>
</tr>
<tr>
<td>exit_on_filter_region</td>
<td>It true, and if an exclusion filter is implemented for this contact behavior, an early exit of the behavior may be allowed when or if the contact moves into a region that would no longer satisfy the exclusion filter. The default it false.</td>
</tr>
<tr>
<td>extrapolate</td>
<td>If true, contact position is extrapolated from last position and trajectory.</td>
</tr>
<tr>
<td>ignore_group:</td>
<td>If specified, the contact group may not match the given ignore group. If multiple ignore groups are specified, the contact group must be different than all ignore groups. Introduced after Release 19.8.1. Section 23.2</td>
</tr>
</tbody>
</table>
ignore_name: If specified, the contact name may not match the given ignore name. If multiple ignore names are specified, the contact name must be different than all ignore names. Introduced after Release 19.8.1. Section 23.2

ignore_region: If specified, the contact group may be in the given ignore region. If multiple ignore regions are specified, the contact position must be external to all ignore regions. Introduced after Release 19.8.1. Section 23.2

ignore_type: If specified, the contact type may not match the given ignore type. If multiple ignore types are specified, the contact type must be different than all ignore types. Introduced after Release 19.8.1. Section 23.2

match_group: If specified, the contact group must match the given match group. If multiple match groups are specified, the contact group must match at least one match group. Introduced after Release 19.8.1. Section 23.2

match_name: If specified, the contact name must match the given match name. If multiple match names are specified, the contact name must match at least one. Introduced after Release 19.8.1. Section 23.2

match_region: If specified, the contact must reside in the given convex region. If multiple match regions are specified, the contact position must be in at least one match region. The multiple regions essentially can together support a non-convex regions. Introduced after Release 19.8.1. Section 23.2

match_type: If specified, the contact type must match the given match type. If multiple match types are specified, the contact type must match at least one match type. Introduced after Release 19.8.1. Section 23.2

on_no_contact_ok If false, a helm error is posted if no contact information exists. Applicable in the more rare case that a contact behavior is statically configured for a named contact. The default is true.

strict_ignore If true, and if one of the ignore exclusion filter components is enabled, then an exclusion filter will fail if the contact report is missing information related to the filter. For example if the contact group information is unknown. The default is true.

time_on_leg The time on leg, in seconds, used for calculating closest point of approach.

Listing 27.3: Configuration Parameters for the Trail Behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nm_radius:</td>
<td>If in this range to contact and ahead of it, slow down. The default is 20 meters.</td>
</tr>
<tr>
<td>no_alert_request:</td>
<td>If true, no request will be made to the contact manager for contact alerts. The default is true.</td>
</tr>
<tr>
<td>post_trail_dist_on_idle:</td>
<td>If true, post TRAIL_DISTANCE even during the idle state. The default is true.</td>
</tr>
<tr>
<td>pwt_outer_dist:</td>
<td>Range to contact outside which the behavior has zero priority.</td>
</tr>
<tr>
<td>radius:</td>
<td>If outside this radius to the contact, head to nm_radius ahead of trail point. The default is 5 meters.</td>
</tr>
<tr>
<td>trail_angle:</td>
<td>Relative angle to the contact to set the trail-point. The default is 180.</td>
</tr>
<tr>
<td>trail_angle_type:</td>
<td>Either &quot;relative&quot; or &quot;absolute&quot; bearing/angle to the contact. The default is relative.</td>
</tr>
</tbody>
</table>
trail_range: Relative distance to the contact to set the trail-point. The default is 50 meters.

Listing 27.4: Example Configuration Block.

```moos
Behavior = BHV_Trail
{
    // General Behavior Parameters
    // ---------------------------
    name = trail_ // example
    pwt = 100 // default
    condition = TRAIL_ALLOWED = true // example
    updates = TRAIL_INFO // example
    templating = spawn // example

    // General Contact Behavior Parameters
    // -----------------------------------
    contact = to-be-set // example
    decay = 15,30 // default (seconds)
    extrapolate = true // default
    on_no_contact_ok = true // default
    time_on_leg = 60 // default (seconds)

    // Parameters specific to this behavior
    // ------------------------------------
    nm_radius = 20 // default (meters)
    no_alert_request = false // default
    post_trail_dist_on_idle = true // default
    pwt_outer_dist = 0 // default (meters)
    radius = 5 // default (meters)
    trail_angle = 180 // default (degrees)
    trail_angle_type = relative // default (or absolute)
    trail_range = 50 // default (meters)
}
```

27.2 Variables Published

The below MOOS variables will be published by the behavior during normal operation, in addition to any configured flags. A variable published by any behavior may be suppressed or changed to a different variable name using the `post_mapping` configuration parameter described in Section 7.2.8.

Listing 27.5: Variables Published by the Trail Behavior.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURSUIT</td>
<td>1 if in behavior is active and producing an objective function, 0 otherwise.</td>
</tr>
<tr>
<td>REGION</td>
<td></td>
</tr>
<tr>
<td>TRAIL_HEADING</td>
<td></td>
</tr>
<tr>
<td>TRAIL_SPEED</td>
<td></td>
</tr>
<tr>
<td>TRAIL_RANGE</td>
<td></td>
</tr>
<tr>
<td>MAX_RANGE</td>
<td></td>
</tr>
</tbody>
</table>
27.3 Parameters

- **trail_range**: The range component of the relative position to the contact to trail.

- **trail_angle**: The relative angle of the relative position to the contact to trail. (180 is directly behind, 90 is a parallel track to the contacts starboard side, -90 is on the port side of the contact.)

- **trail_angle_type**: The trail angle may be set to either relative (the default), or absolute.

- **radius**: The distance (in meters) from the trail position that will result in the behavior “cutting range” to the trail position, and inside of which will result in the behavior “shadowing” the contact. The default is 5 meters.

- **nm_radius**: The distance in meters from the trail point within which the speed will be gradually change from the outer chase speed (max speed) and the speed of the contact, as illustrated in Figure 58. This parameter should typically be set to several times the value of radius to achieve smooth formation flying. Default is 20 meters.

- **max_range**: The distance (in meters) that the contact must be within for the behavior to be active and produce an objective function. The default is max_range value is zero meaning it will be active regardless of the distance to the contact.
The Convoy Behavior

This behavior will drive the vehicle to follow another specified vehicle by following virtual markers dropped by the lead vehicle. The vehicle following the lead vehicle is referred to as the convoying vehicle. As the lead vehicle transits, the convoying vehicle will make note of the lead vehicle’s position and periodically note as markers, the position of the lead vehicle. The convoying vehicle will continuously transit to the rear-most marker. Upon reaching the rear-most marker it will proceed to the next rear-most marker, and so on.

This behavior can be compared to the Trail behavior. The goal of the Trail behavior is to continually aim toward a point at a relative angle and distance from the lead vehicle. When the Trail behavior is using a trail angle of 180 degrees, it may look similar to a Convoy behavior when both vehicles are on a long linear transit. However, as shown in Figure 59, when the lead vehicle turns, the Convoy behavior will continue to follow the trajectory of the lead vehicle, while the Trail behavior will deviate to steer directly toward the point currently 180 degrees behind the lead vehicle.

![Convoy vs. Trail](image)

Figure 59: Convoy vs. Trail When traveling on a linear path, the Convoy behavior is similar to the Trail behavior with the trail angle set to 180 degrees, as shown on the left. But as the lead vehicle maneuvers, the Convoy behavior will follow the path of the lead vehicle, whereas the Trail behavior will continuously aim for the point currently 180 degrees behind the lead vehicle, as shown on the right.

The Convoy behavior may be preferred when it is important to have the following vehicle traverse the same path of the lead vehicle, regardless of their relative angle at any one given instant. For example, if the lead vehicle is driven by a human, or is autonomous but has a more capable sensor system, it may serve as a guide through obstacles, bridge pylons, or other hazards.

28.1 Generation and Removal of Markers

A marker is comprised of the observed X-Y or Lat/Lon position of the lead vehicle at some previous point in time. The lead vehicle typically does not communicate marker information explicitly to the convoying vehicle. Rather the convoying vehicle notes and stores this information from incoming node reports of the lead vehicle. Incoming node reports may be derived via AIS, direct communications, or through sensors on the convoying vehicle. The source of the lead vehicle position information is irrelevant to this behavior.
Figure 60: **Convoy Markers:** The convoying vehicle heads toward the rear-most marker behind the lead vehicle. As the convoying vehicle reaches each marker, the reached marker is dropped, and then proceeds to next rear-most marker as shown on the left. In a lagging situation, shown on the right, the convoying vehicle is still heading toward the rear-most marker, but the convoying vehicle’s path, while it is lagging, may not closely follow the lead vehicle’s path.

The *marker tail* is the set of all markers. The *marker tail length* is the sum of line segments between markers plus the range from the lead vehicle to the *lead marker*, which is the most recently added marker just aft of the lead vehicle. The *aft marker* is the oldest marker in the marker tail and presumably the marker that the convoy behavior is driving toward. The range between markers is set with the configuration parameter `inter_mark_range`. The default value is 10 meters.

A new lead marker is added as the lead vehicle moves, whenever the range between the lead vehicle and the current lead marker becomes greater than the `inter_mark_range`. A marker is removed from the rear of the tail if the tail length becomes longer than `tail_length_max`. The default of this parameter is 150 meters. By this policy, the tail length should always be less than or equal to `tail_length_max`. The rear-most marker may also be removed when or if the convoying vehicle captures the marker. Capturing is discussed next.

### 28.2 Capturing a Marker

As the convoying vehicle approaches its next marker, there are conditions continually examined to determine when it has reached, or *captured*, the marker. The simplest criteria is when the vehicle has reduced its range to the marker to be less than the `capture_radius`, a configuration parameter with default value of 5 meters. This is shown on the left in Figure 61. Occasionally the convoying vehicle may miss the marker, perhaps due to external environmental forces, or other mission considerations that pull the vehicle off a direct path to the marker. If the `capture_radius` were the only criteria for capturing a marker, the convoying vehicle may be forced to loop back to try again. If the `capture_radius` is too large, then the vehicle would be free, perhaps too early, to begin heading toward the next marker. If the lead vehicle is on a highly nonlinear path, a very large `capture_radius` may result in the convoying vehicle deviating too much from the lead vehicle’s path.
The `slip_radius` configuration parameter specifies a larger range to the marker than the `capture_radius`. Once the convoy vehicle enters within range of the `slip_radius`, the marker will be considered captured if the angle between ownship, the marker and next marker, is less than 90 degrees.

The default value for the `slip_radius` is 20 meters. The `slip_radius` range, as a rule of thumb, should be 2-4 times bigger than the `capture_radius`. A configuration warning will be generated if the `slip_radius` is less than the `capture_radius`.

When a marker is captured, the Convoying behavior removes the marker from its internal marker tail. Markers are removed from the tail when either they are captured by the convoying vehicle, or when the marker tail length exceeds the maximal length set by the `max_tail_length` parameter.

### 28.3 The Ideal Convoying Steady State

The ideal, or steady-state, situation for the convoying vehicle is to be positioned, on the marker tail, at the proper convoy range to the lead vehicle, and matching the speed of the lead vehicle. The `convoy range` is the length of the marker tail plus the distance between the rear-most marker and position of the convoying vehicle, as shown in Figure 62.
The ideal convoy range is synonymous with the value given by the parameter \texttt{tail\_length\_max}. The convoy range delta is the difference between the current convoy range and the ideal convoy range. The speed of the convoying vehicle in the ideal convoying steady state, is simply the current speed of the lead vehicle. The convoy speed delta is the difference between the current speed of the convoy vehicle and the lead vehicle.

It is also sometimes useful to have a notion of when the vehicle is in the ideal convoying steady state. As a general approximation we will say it is in the ideal state if:

- the convoy range delta is less than the inter mark range plus the slip radius, and
- the convoy speed delta is less than a quarter of the current lead vehicle speed, and less than a quarter of the convoy vehicle speed.

### 28.4 Speed of the Convoying Vehicle - The Speed Policy

The convoy behavior uses a speed policy to determine which speed best keeps the vehicle at the ideal convoy range to the lead vehicle. The policy is based on (a) the current convoy range, (b) the speed of the lead vehicle, and (c) the direct range between vehicles. The latter is only used as a measure for triggering a safety full-stop. Depending on the current convoy range delta, the speed policy will identify a set speed. The set speed will either match, overshoot, or undershoot the lead vehicle speed depending on whether the current convoy range is ideal, too far, or too close respectively.

When the current convoy range is not ideal, the degree of the overshoot or undershoot in the set speed is linearly proportional to the difference between the convoy range and ideal convoy range. There are several configuration parameters that determine the speed policy. These parameters are conveyed in Figure 63. Note that for the ideal convoy range there is a tolerance. Within this tolerance range, the set speed will simply be the lead vehicle speed, even if the current convoy range is not quite equal to the ideal convoy range.
Figure 63: The Convoy Speed Policy: The preferred speed of the convoy behavior is determined by the speed policy, which depends on (a) the speed of the lead vehicle, (b) the current convoy range, and (c) the current direct range between vehicles. The speed policy is configured by the user and may be dynamically adjusted during a mission. There are six correction modes identified, based on the relationship between the current convoy range and the ideal convoy range.

The Correction Mode:

The relationship between the current convoy range and the ideal convoy range determines which of six correction modes apply at the moment. They are shown in Figure 63, and named:

1. **full stop**: The current convoy range is less than or equal to the `full_stop_convoy_range`.
2. **close**: The current convoy range is greater than the `full_stop_convoy_range`, but less than or equal to the `slower_convoy_range`.
3. **ideal close**: The current convoy range is greater than the `slower_convoy_range`, but less than or equal to the `ideal_convoy_range`.
4. **ideal far**: The current convoy range is greater than the `ideal_convoy_range`, but less than or equal to the `faster_convoy_range`.
5. **far**: The current convoy range is greater than the `faster_convoy_range`, but less than or equal to the `full_lag_convoy_range`.
6. **full lag**: The current convoy range is greater than the `full_lag_convoy_range`.

### 28.4.1 Speed Policy Configuration Parameters

The speed policy is set directly with the following five parameters as an example. Note the value of the ideal convoy range does not directly influence the speed policy. Speed adjustments faster or slower are only made when the current convoy range is outside the steady speed span. The `ideal_convoy_range` parameter is used in setting the correction mode, and to have a metric to measure mission performance against. If left unspecified, it will be automatically set to the midpoint between the `faster_convoy_range` and the `slower_convoy_range`.

- `full_stop_convoy_range=20`
- `slower_convoy_range=40`
- `ideal_convoy_range=50`
faster_convoy_range=60
full_lag_convoy_range=80

From Figure 63, the legal relative values are conveyed. For example, the slower_convoy_range cannot be less than the full_stop_convoy_range. The relative magnitude for each adjacent pair of parameters is checked both on helm startup, and after any dynamic update to the behavior during a mission. If a violation is detected upon startup, the helm will start in the malconfig state and will be unable to deploy until the behavior file is fixed and the helm re-launched. If the violation is detected at runtime, the behavior update will simply be rejected and a run warning will be posted.

28.4.2 Speed Policy Compression

The speed policy, comprised of the configuration parameters described above, may be dynamically adjusted mid-mission to compress or expand the convoy range between vehicles. This could be done using the updates variable, by providing new values for the speed policy parameters. For example, if the updates variable were CONVOY_UPDATES, then the following posting would adjust the speed policy:

CONVOY_UPDATES = full_lag_convoy_range=50 # faster_convoy_range=40 # slower_convoy_range=30

While the above is supported, a perhaps easier method is to specify a single compression value. This parameter ranges between 0 (no compression), and 1 (full compression). Since a fully compressed policy would essentially amount to a policy comprised of either full catch up speed or full stop, compression is capped at 0.9. For example, given the example speed policy above in Section 28.4.1, the full CONVOY_UPDATES line above could be accomplished with the simpler line:

CONVOY_UPDATES = compression=0.5

Compression is applied to all speed policy intervals except the safety-critical full_stop_convoy_range. The idea is conveyed in Figure 64 below.
Figure 64: Speed Policy Compression: A single compression value in the range of [0,0.9] may be applied to a compression policy. All intervals will be compressed by this value, except the safety-critical full_stop_convoy_range.

Note that by setting compression=N either on startup, or through the updates variable, the compression is applied to the original uncompressed policy. In other words, two successive updates with a compression value of 0.5, will result in a policy compressed at 0.5, not 0.25.

28.4.3 Off-Peak Speed Preferences

The convoy behavior, like all behaviors of the IvP helm, will produce an objective function. The objective function of the convoy behavior is defined over possible course (desired heading) and speed choices. The convoy behavior constructs this 2D objective function by combining an objective function (utility function) defined over speed and one defined over heading. In this section the speed utility function is discussed.

The helm is configured with a decision space over all decision variables. This decision space is propagated to all behaviors of the helm, including the convoy behavior. The decision space for the speed component will contain both a min speed value (typically zero) and a max speed value. The max speed may be the max speed of the vessel, or the highest allowed operational speed, or may be set for some other reason. In any event, the speed utility function maps speed to utility, from the min speed to the max speed. As shown in Figure ??, the peak of this function is the set speed, produced from the speed policy. The set speed and speed utility function are dynamic, recalculated at each iteration of the helm based on situational input.
Figure 65: **Speed Utility Function**: Given a set speed, the speed utility function is marked by the optimal utility at the set speed with linearly decreasing utility for speed differing from the set speed. Note in Figure 65, by default, the utility at both zero speed and the maximum speed are set to zero. However, the convoy behavior will vary either the zero speed or max speed utility to a non-zero value depending on the correction mode, described in Section 28.4. For example, Figure 66 shows the speed utility function when the correction mode is **far**. Since the current convoy range is larger than the ideal convoy range, the correction mode is **far**, and the set speed already is somewhat higher than the lead vehicle speed, to close the convoy range. However, in this correction mode, off-peak utility of speeds higher than the set speed are skewed higher by setting the utility of the max speed to 50 percent of the max utility. In this way, if the convoy behavior needs to coordinate and compromise with another behavior, it is more likely to strike a compromise with a speed higher than the set speed.

Figure 66: **Speed Utility Function When Lagging**: In a lagging situation, the set speed is chosen accordingly to close range on the lead vehicle. Additionally, the utility of higher speeds degrade linearly at a slower rate than the utility of speeds lower than the set speed. If the behavior is in a compromise situation with another behavior and the optimal set speed is precluded by the other behavior, compromise speeds are more likely to be higher than the set speed.

Likewise, if the convoy vehicle has a correction mode of **close**, Figure 67 shows the speed utility function. Since the current convoy range is smaller than the ideal convoy range, and the correction mode is **close**, the set speed already is somewhat lower than the lead vehicle speed, to open the convoy range. However, in this correction mode, off-peak utility of speeds lower than the set speed...
are skewed lower by setting the utility of the zero speed to 50 percent of the max utility. In this way, if the convoy behavior needs to coordinate and compromise with another behavior, it is more likely to strike a compromise with a speed lower than the set speed.

Figure 67: Speed Utility Function When Close-In: In a close-in or tailgating situation, the set speed is chosen accordingly to open range to the lead vehicle. Additionally, the utility of slower speeds degrade linearly at a slower rate than the utility of speeds higher than the set speed. If the behavior is in a compromise situation with another behavior and the optimal convoy set speed is precluded by the other behavior, compromise speeds are more likely to be slower than the set speed.

The speed utility function is skewed in this manner, based on the correction mode as described in the two examples above. For the other correction modes, the following policy is used:

<table>
<thead>
<tr>
<th>Correction Mode</th>
<th>Zero-Speed Utility</th>
<th>Max-Speed Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>ideal_close</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>ideal_far</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>far</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>full lag</td>
<td>0</td>
<td>75</td>
</tr>
</tbody>
</table>

28.5 Visual Preferences

The primary visual artifact produced by the convoy behavior is a posting to MOOS variable VIEW_POINT, for representing the markers in the marker tail. This variable is recognized by the pMarineViewer GUI application. An example posting is:

```
VIEW_POINT = x=-5.1,y=-50.31,label=ben_deb_14,vertex_color=red,vertex_size=6
```

The label is formed from the convoy vehicle name plus the lead vehicle name plus an index. As markers are captured and deleted by the behavior they are "erased" by re-posting with the same label, setting the field active=false.

```
VIEW_POINT = x=-5.1,y=-50.31,active=false,label=ben_deb_14
```
An example of the artifacts rendered in *pMarineViewer* in a typical mission is shown in Figure 68.

![Image of Convoy Behavior Visual Artifacts](image)

Figure 68: **Convoy Behavior Visual Artifacts**: The Convoy behavior will post points of a chosen size and color to represent the working marker tail. The aft marker is typically rendered in white. In this case the marker colors are set to automatically match the vehicle color for extra clarity.

The marker size and color, and the marker label color, may be set with configuration parameters:

- `visual_hint = marker_color = green`
- `visual_hint = marker_label_color = white`
- `visual_hint = marker_size = 6`

The default marker color is *blue*. The default marker label color is *off*. The default marker point size is 4.

### 28.6 Output of the Convoy Behavior in the Form of Publications

All behaviors produce two kinds of output, (a) an objective function, and (b) one or more publications to MOOS variables. The former was discussed in Section ??, the latter is discussed here. MOOS variable publications take two forms, (a) publications that are baked into the code that are not user configurable, and (b) user configurable publication based on event flags. The tendency in newer behaviors is to use fewer baked in publications and instead support a rich set of event flags that are further configurable with event macros. The specifics of publications for the Convoy behavior are discussed here.

#### 28.6.1 Baked In Publications of the Convoy Behavior

Baked in publications are part of a behavior’s implementation and the format of these publications are not configurable by the user. The Convoy behavior has three core publications:

- **VIEW_POINT**: Intended to be consumed by *pMarineViewer* during a mission, or *aloview* during mission playback. This variable was discussed in detail in Section 28.5.
- **CONVOY_SPD_POLICY**: A description of the prevailing speed policy.
- **CONVOY_STAT_RECAP**: A recap of convoy configuration state that typically does not change between iterations. See the example below.
- **CONVOY_RECAP**: A recap of convoy state that changes often and includes the core performance metrics and other state variables. See the example below.

An example of the static recap publication:

```
CONVOY_STAT_RECAP = follower=abe,leader=deb,ideal_rng=25,compression=0,index=0
```

An example of the speed policy publication:

```
CONVOY_SPD_POLICY = full_stop_rng=2,vname=eve,slower_rng=23,ideal_rng=25, faster_rng=27,full_lag_rng=40,lag_spd_delta=2, max_compress=0.9
```

An example of the dynamic recap publication:

```
CONVOY_RECAP = convoy_rng=17.4,vname=gus,rng_delta=-7.5,tail_cnt=6, cmode=close,tail_rng=4.5,tail_ang=3.44,mark_bng=46.41, almnt=49.84,set_spd=0.661,cnv_avg2=0.905,almnt=0.867, mx=30.6,my=-11.8,mid=0
```

The `mx`, `my`, and `mid` components are the aft marker X, Y location and marker ID respectively. The `cmode` component is the most recently observed correction mode.

Although the format of these variables cannot be changed in the mission file, they can be remapped to a different variable name, or they may be suppressed if desired, using the IvP Behavior `post_mapping` parameter:

```
post_mapping = CONVOY_RECAP,silent
post_mapping = CONVOY_RECAP,MY_PREFERRED_RECAP_VAR
```

By default, the `CONVOY_RECAP` will be posted whenever the behavior drops the aft marker, typically by reaching it. If the user prefers a more frequent posting, on every iteration of the behavior, the following parameter is used:

```
post_recap_verbose=true
```

The default setting for this parameter is false.

If the preference is to publish a smaller subset of the recap, or a simple numerical posting more readily consumable for plotting, use the event macros instead. For example, if one only wants the alignment value (the `almnt` field in the recap), an event flag can be used:

```
convoy_flag = CONVOY_ALIGNMENT=${ALIGNMENT}
```

which is published on every iteration of the convoy behavior when active. Or for less frequent publications, whenever a marker is reached:

```
marker_flag = CONVOY_ALIGNMENT=${ALIGNMENT}
```

See Section 28.6.3 for further event flag and macro options.
28.6.2 Event Flag Publications of the Convoy Behavior

Event Flags for All IvP Behaviors

The Convoy behavior is a subclass of the general IvP Behavior class and thus inherits the event flags of the base class. These event flags are documented outside this scope, but the list of event flags is: activeflag, endflag, idleflag, inactiveflag, runflag, spawnflag.

Event Flags for All IvP Contact Behaviors

The Convoy behavior is a subclass of the general IvP Contact Behavior class, which in turn is a subclass of the IvP Behavior class. A contact behavior inherits the cnflag event flag. This single event flag supports several types of events that, for example, can be triggered based on the range to the contact, crossing the contact from port to bow, or crossing passing the contact from fore to aft, or upon closest point of approach (CPA) to the contact, among several others.

28.6.3 Event Flag Macros of the Convoy Behavior

Event Flag Macros for All IvP Behaviors

The Convoy behavior is a subclass of the general IvP Behavior class and thus inherits the event flag macros of the base class. These event flags are documented outside this scope, but the list of event flag macros is: OWNSHIP, BHVNAME, BHVTYPE, PWT, UTC, OSX, OSY, OSV, CTR, CTR1, CTR2, CTR3, CTR4, CTR5.

Event Flag Macros for All IvP Contact Behaviors

The Convoy behavior is a subclass of the general IvP Contact Behavior class and thus inherits the event flag macros of this base class. These event flags are documented outside this scope, but the list of event flag macros is: CNX, CNY, CNH, CNV, CN_NAME, CN_VTYPE, CN_RANGE, RANGE, ROC, DS_CN_REL_BNG, CN_OS_REL_BNG, BNG_RATE, CN_SPD_IN_OS_POS, OS_FORE_OF_CN, OS_AFT_OF_CN, OS_PORT_OF_CN, OS_STAR_OF_CN, CN_FORE_OF_OS, CN_AFT_OF_OS, CN_PORT_OF_OS, CN_STAR_OF_OS.

Listing 28.6: Macros supported by the Convoy Behavior.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG_SPD2</td>
<td>The average speed of the lead vehicle over the last previous two seconds.</td>
</tr>
<tr>
<td>AVG_SPD5</td>
<td>The average speed of the lead vehicle over the last previous five seconds.</td>
</tr>
<tr>
<td>COMP</td>
<td>The current compression setting. See Section 28.4.2.</td>
</tr>
<tr>
<td>CONVOY_rng</td>
<td>The current convoy range. See Section 28.3 for the definition of the convoy range.</td>
</tr>
<tr>
<td>IDEAL_rng</td>
<td>The current ideal convoy range. See Section 28.3 for the definition of the ideal convoy range.</td>
</tr>
<tr>
<td>LEADER</td>
<td>The name of the leader vehicle, in lower case.</td>
</tr>
<tr>
<td>CMODE</td>
<td>The correction mode for the most recent set speed calculations. See Section 28.4 for the definition of the correction mode.</td>
</tr>
<tr>
<td>SET_SPD</td>
<td>The current set speed. See Section 28.4 for the definition of the set speed.</td>
</tr>
<tr>
<td>RECAP</td>
<td>A summary of the Convoy behavior settings and state.</td>
</tr>
<tr>
<td>TAIL_CNT</td>
<td>The number of markers currently in the marker tail. See Section 28.1 for the definition of the marker tail.</td>
</tr>
</tbody>
</table>
TRK_ERR: The current range to the marker tail augmented with recent history of retired markers. Meant to measure the deviation from the path of the lead vehicle. See Figure 70 for the definition of the track error.

An example of the RECAP macro expansion is:

$$\text{RECAP} = \text{leader=ben, follower=abe, convoy\_range=32.1, ideal\_range=30, compression=0.25}$$

28.7 Performance Metrics

Performance metrics are values derived from the current ownship and contact positions and headings, plus the current marker tail and the parameter set by the user for the ideal contact range.

1. tail range: The range between ownship and the aft marker.
2. tail angle: The angle between the three points given by ownship position, the aft marker and the near-aft marker. By definition this value cannot be more than 180. This number is reported at 180-x so that minimal, near zero is optimal. See Figure 69.
3. marker bearing: The absolute relative bearing of the aft marker to ownship. This number is in the range of [0, 180]. See Figure 69.

![Diagram showing tail angle and marker bearing](image)

Figure 69: Tail Angle and Marker Bearing: The tail angle is angle at the aft marker between ownship and the near-aft marker. The marker bearing is the absolute relative bearing of the aft marker to ownship.

4. alignment: Alignment is the sum of tail angle and marker bearing.
5. convoy range delta: The difference between the current convoy range and the ideal convoy range.
6. track error: The distance to the marker tail with the marker tail augmented by a few recently retired markers. This is meant to convey how well ownship is following the lead vehicle. See Figure 70.
Figure 70: **Track Error**: The track error is the distance between ownship and the track line segments, formed by the marker tail and few recently retired markers, ghost markers.

The metrics are published in the **CONVOY_RECAP** posting as discussed in Section 28.6.1. They may also be published via event flags and macros as discussed in Section 28.6.3.

### 28.8 Configuration Parameters

Certain configuration parameters available to Convoy behavior are unique to the Convoy behavior and others are inherited one or more base classes. Here all parameters are presented, with details on those unique to the Convoy behavior.

#### Parameters for all IvP Behaviors

The Convoy behavior is a subclass of the general IvP Behavior class and thus inherits substantial capabilities and parameters for configuring the base class functionality. These parameters are documented outside this scope, but the list of parameters is: `activeflag, condition, duration, duration_idle_decay, duration_reset, duration_status, endflag, idleflag, inactiveflag, name, nostrave, perpetual, post_mapping, priority, pwt, runflag, spawnflag, templating, updates`.

#### Parameters for all IvP Contact Behaviors

The Convoy behavior concerns a contact (the lead vehicle), and shares common functionality with several other behaviors dealing with a contact. As such, the Convoy behavior is a subclass of the general IvP Contact Behavior which in turn is a subclass of the IvP Behavior class and thus inherits substantial capabilities and parameters for configuring the functionality of the two base classes. These parameters are documented outside this scope, but the list of parameters is: `bearing_lines, contact, decay, extrapolate, match_group, match_name, match_region, match_type, ignore_group, ignore_name, ignore_region, ignore_type, on_no_contact_ok, strict_ignore, exit_on_filter_group, exit_on_filter_vtype, exit_on_filter_region, time_on_leg`.

#### Parameters for the Convoy Behavior

*Listing 28.7: Configuration Parameters for the Convoy Behavior.*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>capture_radius</strong></td>
<td>The range between the convoying vehicle and the next marker before the marker can be considered captured. Section 28.2.</td>
</tr>
</tbody>
</table>
**compression**: The degree to which the ideal convoy range and the entire speed policy is compressed to achieve closer convoy ranges. This parameter ranges between \([0, 0.9]\). Section 28.4.2.

**faster_convoy_range**: The convoy range above which the behavior will produce preferred speeds that are faster than the lead vehicle speed. Section 28.4.

**full_stop_convoy_range**: Range between ownship and the lead vehicle, within which the preferred ownship speed is zero. Section 28.4.

**full_lag_convoy_range**: The convoy range above which ownship will apply maximum speed in order to close the convoy range. Maximum speed is given by the lead vehicle speed plus the lag_speed_delta. Section 28.4.

**inter_mark_range**: The range between neighboring markers. Section 28.1.

**ideal_convoy_range**: The ideal or target convoy range to be achieved by the behavior. Section 28.4.

**radius**: Same as capture_radius.

**lag_speed_delta**: The difference between preferred ownship speed and the lead vehicle speed when the convoy range is at or higher than the full_lag_convoy_range. Section 28.4.

**slip_radius**: The range between the convoying vehicle and the next marker, within which the marker will be considered captured when or if the vehicle begins to open range to the marker. Section 28.2.

**slower_convoy_range**: The convoy range within which the behavior will produce preferred speeds that are slower than the lead vehicle speed. Section 28.4.

**tail_length_max**: The maximum marker tail length, beyond which the behavior will drop the oldest marker. Section 28.1.

**visual hints**: One or more hints governing the preferred size and color of markers. Section 28.5.

*Listing 28.8: Example Configuration Block.*
Behavior = BHV_ConvoyV21
{
    // General Behavior Parameters
    name = convoy_
pwt = 100
    condition = MODE = convoying
    updates = CONVOY_UPDATES
    templating = spawn

    // Contact Behavior Parameters
    contact = unset_ok

    // Convoy Behavior Parameters
    radius = 3.0
    slip_radius = 15.0
    inter_mark_range = 3
    tail_length_max = 40
    full_stop_convoy_range = 2
    slower_convoy_range = 23
    ideal_convoy_range = 25
    faster_convoy_range = 27
    full_lag_convoy_range = 40
    lag_speed_delta = 2.0

    marker_flag = CONVOY_RECAP=${RECAP}
    marker_flag = CONVOY_RANGE=${CONVOY_RNG}
    visual_hints = marker_color=$(COLOR)
    visual_hints = marker_color_label=off
    visual_hints = marker_size=12
}

28.9 Terms and Definitions

- **aft marker**: The oldest marker, typically farthest to the lead vehicle of all the markers, and typically the marker currently being driven toward by the Convoy behavior. Section 28.1.
- **captured marker**: The marker reached by the following vehicle. Section 28.1.
- **convoy range**: The actual current distance along the marker tail from the lead vehicle to the following vehicle. Section 28.3.
- **compression**: Modification of the speed policy to adjust all convoy range parameters, except the full stop convoy range, to be smaller than the original speed policy values. Section 28.4.
- **convoy range delta**: The absolute difference between the ideal convoy range and the desired convoy range. Section 28.3.
- **convoy speed delta**: The absolute difference between the lead vehicle speed and the following vehicle speed. Section 28.3.
- **ideal convoy range**: The desired convoy range. Section 28.3.
- **lagging**: A following vehicle is lagging the lead vehicle when the convoy range is beyond the ideal convoy range, namely beyond the range set in the faster_convoy_range. Section 28.4.
- **lead marker**: The most recently created marker, typically closest to the lead vehicle of all the markers. Section 28.1.
• **marker**: A point in the X-Y plane representing a prior location of the lead vehicle, to be driven toward by the convoying or following vehicle. Section 28.1.

• **marker tail**: The set of markers used for determining the following vehicle’s path to match the lead vehicle. Section 28.1.

• **marker tail length**: The distance between the lead vehicle and the newest marker plus the total length of segments between markers. Section 28.1.

• **range**: The direct range between the lead and following vehicle.

• **set speed**: The speed chosen to modify or hold the convoy actual convoy range compared to the ideal convoy range. It is the output of the speed policy. Section 28.4.

• **speed policy**: The policy governing the selection of the set speed given the current convoy range, lead vehicle speed, and absolute direct range between vehicles. Section 28.4.

• **track error**: The distance between ownship and the marker tail, where the marker tail is augmented by a few recently retired ghost markers. See Figure 70.

### 28.10 Known Shortcomings - Work in Progress

- Automatic compression. Speed policy compresses as the lead vehicle slows, and decompresses as the lead vehicle speeds up.
- Name of the leader is a macro option
- Follow auto-informs heartbeat style to the leader
29 Extended Example Missions with the IvP Helm

This section addresses a number of example mission configurations the user may run in simulation, to explore various aspects of the helm, certain behaviors, and certain tools for configuring, running and analyzing missions. All missions are available in the moos-ivp software tree in the following location:

```
$ cd moos-ivp/ivp/missions/
```

Obtaining the moos-ivp software tree from the public server was discussed in Section 4.

29.1 Preliminaries

Each mission in the `missions/` directory is contained in a dedicated subdirectory such as `s1_alpha`, or `m2_berta`. The 's' and 'm' in the prefix indicate whether it is a single-vehicle or multi-vehicle mission. The numbering roughly correlates to an increase in complexity from commonly used bread-and-butter behaviors and configurations to more complex mission configurations.

Each directory contains one or more mission files, i.e., files that end in the `.moos` suffix, and one or more behavior files, i.e., files that end in the `.bhv` suffix. During the course of running a mission, certain additional temporary files may be created. In all cases, the temporary files may be removed by running the `clean.sh` script included in each directory. The configuration and launching of the missions requires some familiarity with three tools, (a) pAntler, for launching multiple MOOS applications, (b) the `nsplug` tool, for building composite `.moos` and `.bhv` files from a template and set of include files, and (c) basics of shell scripts for organizing the overall launch process and cleanup. A brief recap of these three tools is covered next.

29.2 Using pAntler to Launch Missions

The `pAntler` utility is a command-line tool used to orchestrate the launching of a group of MOOS processes. A typical invocation is:

```
$ pAntler charlie.moos
```

The MOOS file contains a configuration block for each process, and an Antler configuration block which is typically the first block in the file. Each block contains a collection of local “parameter = value” pairs read in by each individual application. There are also typically a number of global parameter=value pairs for possible use by all applications.

29.3 Using the nsplug Utility for Configuring Mission and Behavior Files

In multi-vehicle missions, the maintenance of mission and behavior files across multiple vehicles can become cumbersome and user error-prone. Since many of the file components are identical between vehicles, a fair amount of the problem is mitigated by building the files from file template files that include the common components from separate files. The `nsplug` command-line utility is used to expand a template file into a target file, with arguments passed that typically distinguish
the produced file from others produced from the same template. For example, two mission files, one for the vehicle "charlie", and one for the vehicle "frankie" may differ only in their vehicle names and IP Port number, and may be produced from a template file with the following two commands:

```
$ nsplug meta_vehicle.moos targ_charlie.moos VNAME=charlie PORT=9201
$ nsplug meta_vehicle.moos targ_frankie.moos VNAME=charlie PORT=9202
```

For the multi-vehicle example missions distributed with the moos-ivp tree and described in this section, this use of templates is used in all cases. The mission directories downloaded will not contain the `nsplug" output" files such as `targ_charlie.moos` above. Rather, they are built by executing the launch scripts provided in each directory with a command such as:

```
$ ./launch.sh
```

The above invocation will both build the mission and behavior files and invoke pAntler for each vehicle to start the simulation. Certain common command-line options such as `./launch --just_build", to build the mission files without launching, are available in all scripts. A few conventions with respect to templates and `nsplug` are used across all example mission directories.

1. Files used as templates have a "meta" prefix.
2. Files meant to be included or plugged into another template file have a "plug" prefix.
3. Output files that are built from templates and plug-ins via `nsplug` have "targ" prefix.
4. Template or plug-in files that contribute to a target MOOS file have ".moos" suffix.
5. Template or plug-in files that contribute to a target Helm behavior file have ".bvh" suffix.

For example, in the `m2_berta` mission, the following files exist prior to building any target files and launching:

```
$ cd missions/m2_berta
$ ls
  clean.sh*   plug_uSimMarine.moos   plug_pMOOSBridgeV.moos
  launch.sh*  plug_origin_warp.moos plug_pMarinePID.moos
  meta_shoreside.moos plug_pBasicContactMgr.moos plug_pNodeReporter.moos
  meta_vehicle.bhv  plug_pHelmIVP.moos  plug_uProcessWatch.moos
  meta_vehicle.moos  plug_pLogger.moos  plug_uXMS.moos
```

After building all the target files, the following additional `targ.*` files exist:

```
$ ./launch.sh --just_build
$ ls
  clean.sh*         plug_pBasicContactMgr.moos   plug_uXMS.moos
  launch.sh*        plug_pHelmIVP.moos           targ_gilda.bhv
  meta_shoreside.moos plug_pLogger.moos         targ_gilda.moos
  meta_vehicle.bhv   plug_pMOOSBridgeV.moos      targ_henry.bhv
  meta_vehicle.moos  plug_pMarinePID.moos       targ_henry.moos
  plug_uSimMarine.moos plug_pNodeReporter.moos  targ_shoreside.moos
  plug_origin_warp.moos plug_uProcessWatch.moos
```

232
The mission may be launched, with MOOSTimeWarp=10, by:

$ ./launch.sh 10
30  The Charlie Mission

**Behaviors:**  Loiter, StationKeep, Waypoint

**MOOS Apps:**  pHelmIVP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer, uTimerScript

**Primary Topics:**
1. Hierarchical Mode Declarations
2. The Loiter behavior
   2a. Loiter traversal - Clockwise vs. counter-clockwise
   2b. Loiter polygon shapes, hexagons, ellipses
   2c. Loiter dynamic changes to the polygon center position
   2d. Loiter robustness to periodic external forces
3. The StationKeep Behavior

**Side Topics:**
1. uTimerScript is used to simulate wind gusts, external forces
2. Use of pMarineViewer action buttons and action pull-down menu

30.1 Launching the Charlie Mission

The charlie mission may be launched from the command line in the following manner:

```
$ cd moos-ivp/missions/s3_charlie/
$ ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 72, with a single vehicle, *henry*, initially in the PARK mode. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "LOITERING" mode and begins to proceed to the polygon shown. To get a quick feel for what’s possible in this simulation, the following are some things to try. (a) Click on the "CWISE" and "CCWISE" buttons to change the direction of loitering around the polygon. (b) Click the RETURN and DEPLOY buttons repeatedly to see the vehicle switch between two primary modes. (c) Select the "WIND_GUSTS=true" option in the ACTION pull-down menu to see the vehicle adjust to random periodic external forces. (d) Select the "STATION_KEEP=true" option in the ACTION pull-down menu to see how the vehicle station keeps with simulated wind gust. (e) Let the vehicle return to its return point and watch how it automatically switches to the station keeping mode, and then re-deploy it. (f) Select the ellipse polygon from the ACTION pull-down menu to see a different loiter shape. (g) Select the different center points for the loiter ellipse from the ACTION pull-down menu to see just the location of the loiter polygon change and the vehicle adjust.

30.2 Topic #1: Hierarchical Mode Declarations in the Charlie Mission

The Charlie mission is organized by hierarchical mode declarations into four modes: INACTIVE, LOITERING, RETURNING, and STATION-KEEPING. The mode declarations are given at the top of the charlie.bhv file and shown again in Figure 71. The pMarineViewer application by default shows the vehicle’s mode alongside the vehicle name, and when the simulation is first launched, the Charlie vehicle is shown on the screen with the mode PARK.
Figure 71: Charlie Mission Mode Declarations: The helm, when in drive, will be in one of the four modes indicated by the leaf nodes on the tree. The hierarchical mode structure means, for example, that when the vehicle is in the "RETURNING" mode, it is also considered to be in the "ACTIVE" mode. The vehicle mode, by default, is displayed alongside the vehicle name in the pMarineViewer window.

The "PARK" mode displayed with the vehicle when the simulation is launched is actually the high level helm state. As described in Section 5.2, the helm has a higher level helm state which is either PARK or DRIVE, analogous to a car in either the "Park" or "Drive" modes. When the Charlie mission is first launched, the helm is not yet in drive, and is put into drive when the DEPLOY button is hit. This button is configured in the pMarineViewer configuration block to make several MOOS pokes with a single button click. It posts MOOS_MANUAL_OVERRIDE=false, to put the vehicle in the DRIVE helm state. It posts DEPLOY=true to put the helm in the ACTIVE mode, and it posts LOITER=true to put the helm in the ACTIVE:LOITERING mode.

The Charlie mission is configured to allow for easy transition between the other modes via the pMarineViewer interface. The vehicle may be put into the STATION-KEEPING mode at any time via the STATION_KEEP=true option in the ACTION pull-down menu in pMarineViewer. This does nothing more than make the STATION_KEEP=true post to the MOOSDB. The same could have been accomplished by uPokeDB for example:

```bash
$ cd moos-ivp/missions/s3_charlie/
$ uPokeDB charlie.moos STATION_KEEP=true
```

Referring to the mode declarations in Figure 71, the LOITERING and RETURNING modes both have the condition STATION_KEEP != true, so both of those mode requirements fail to be satisfied. The STATION-KEEPING mode is the default mode, i.e., the "else" mode when the requirements of the RETURNING mode are not met. The STATION_KEEP=true posting could also have been made by another MOOS process, for example, connected to an acoustic modem, an Iridium satellite, or wifi interface.

The helm mode is communicated to pMarineViewer from the helm via the MOOS variable IVPHELM_SUMMARY. This variable is parsed and the mode is rendered next to the vehicle name in the viewer. This rendering may be turned off by toggling through the "name options", with the ‘n’ key,
or setting the `vehicles_name_viewable` parameter in the `pMarineViewer` configuration block to one of the values described in the "Vehicles Pull-Down Menu" Section of [5].

### 30.3 Topic #2: The Loiter Behavior

The LOITERING mode in the Charlie mission is characterized by the `BHV_Loiter` behavior, and we explore some of capabilities and options for this behavior here. The behavior configuration, in file `charlie.bhv` is shown in Listing 1 below. The first configuration block, in lines 3-6, are for parameters defined generally for all IvP Helm behaviors, and the second block, in lines 8-16, are for parameters defined explicitly for the Loiter behavior.

**Listing 30.1: Configuration of the Loiter behavior in the Charlie mission, from the file charlie.bhv.**

```plaintext
1 //------------------------------
2 Behavior = BHV_Loiter
3 {
4    name = loiter
5    priority = 100
6    condition = MODE==LOITERING
7    updates = UP_LOITER
8
9    speed = 1.3
10    clockwise = false
11    capture_radius = 4.0
12    nm_radius = 15.0
13    polygon = format=radial, x=0, y=-75, radius=40, pts=6, snap=1,
14    visual_hints = nextpt_color=white, nextpt_lcolor=khaki
15    visual_hints = edge_color=blue, vertex_color=yellow
16    visual_hints = edge_size=1, vertex_size=2, label=LOITER_POLYGON
17 }
```

When the vehicle is first deployed, it heads to the hexagon shaped polygon shown in Figure 72, configured in line 13. It traverses the polygon in a counter-clockwise manner due to the configuration in line 10. The Loiter behavior maintains an internal mode, the "acquire mode" which is true when the vehicle is not sufficiently close to the polygon, defined by the `acquire_dist` parameter which is by default 10 meters. It also keeps track of how many vertex arrivals come by way of achieving the capture radius, and how many by way of the non-monotonic radius. These internal state variables are summarized by the Loiter behavior by publishing a variable `LOITER_REPORT`. An example might look like:

```plaintext
LOITER_REPORT = "index=2,capture_hits=12,nonmono_hits=3,acquire_mode=false"
```

In the Charlie mission, `pMarineViewer` is configured to scope on this variable by default, and the evolving `LOITER_REPORT` may be monitored as the vehicle progresses around the polygon.

#### 30.3.1 Loiter Traversal - Clockwise vs. Counter-Clockwise

The traversal direction of the Loiter behavior is by default clockwise and can be set to counter-clockwise with the parameter `clockwise=false`. In the Charlie mission, the traversal direction can be changed by selecting `UP_LOITER = clockwise=true` and `UP_LOITER=clockwise=false` from the ACTION
pull-down menu. This is a good way to observe the algorithm used by the behavior to acquire the polygon when sufficiently far inside the polygon. The traversal direction could also have been affected by any other MOOS application capable of executing the above two MOOS pokes, including uPokeDB, or any application connected to a communications device.

### 30.3.2 Loiter Polygon Shapes, Hexagons, and Ellipses

The only restriction on the shape traversed by the Loiter behavior is that it be a convex polygon. The typical specification of the polygon is shown in line 13, via the ”radial” syntax, which accepts a center position ("x=0, y=-75"), a distance from the center point to each vertex ("radius=40"), the number of points in the polygon, ("pts=6"), and a snap value, which rounds the vertex positions to, in this case, the nearest whole meter ("snap=1"). Ellipse shapes are supported with a string such as:

```
polygon = format=ellipse, x=110, y=-75, degs=150, pts=14, major=80, minor=20
```

The above ellipse specification is selectable in the Charlie mission via the ACTION pull-down menu. If selected while in the loiter mode, the vehicle immediately begins traversing to this new polygon, as shown in Figure 73.

### 30.3.3 Loiter Dynamic Changes to the Polygon Position

In addition to the an outright change to the loiter polygon as described above, which includes shape changes, number of vertices and position, there are a few methods for just changing the polygon position while leaving the other characteristics in tact. The Loiter behavior accepts a parameter for just the center position. The following two such assignments are selectable in the ACTION pull-down menu in the pMarineViewer for the Charlie mission:

```
center_assign = 40,-40
center_assign = x=100, y=-80
```

Try selecting either of these options, with the polygon set to either the hexagon or ellipse and confirm that only the position changes. One other method for affecting the polygon position is via the center activate parameter, which is false by default. When set to true, the polygon center is set to be the vehicle’s present position when the Loiter behavior enters the running state. In the Charlie mission, try selecting center activate=true from the ACTION pull-down menu. If already in the LOITERING mode, then nothing happens immediately. If and when the helm exits and returns to the LOITERING mode, for example after clicking the RETURN button, and then the DEPLOY button again, note that the loiter polygon is centered on the vehicle’s present position. The center activate feature is particularly useful when the Loiter behavior is used more or less as a station keeping behavior and one wants to be able to command the vehicle to loiter at the present position at any given time.

### 30.3.4 Loiter Robustness to Periodic External Forces

When the Loiter behavior is active and the vehicle is proceeding around the polygon vertices, it is proceeding more or less like the Waypoint behavior (both behaviors share the same WaypointEngine
class as a sub-component). Things get interesting when the vehicle is approaching the polygon from the outside or from a distance internally (for example when the center_activate parameter is set to true). To test and demonstrate this robustness, the Charlie mission is configured with the uTimerScript utility to generate periodic random forces to simulate wind gusts, to push the vehicle off the loiter polygon in unpredictable ways. An example is shown in Figure 74.

The simulated wind gusts may be activated by selecting the WIND_GUSTS=true option from the pMarineViewer ACTION pull-down menu. The details of the simulated forces can be found in the uTimerScript configuration block in the charlie.moos file. The script works by posting external force vectors to the MOOS variable USM_DRIFT_VECTOR_ADD, which is read by the uSimMarine application to alter the prevailing external force vector, which by default has a magnitude of zero. The syntax of the uTimerScript script to implement the drift effects in this example are described in more detail in the "A Script as a Proxy for Simulating Random Wind Gusts" Section of the uTimerScript documentation, [27].

30.4 Topic #3: The StationKeep Behavior

The STATION-KEEPING mode in the Charlie mission is characterized by the BHV_StationKeep behavior, and we explore some of its capabilities and options for this behavior here. The behavior configuration, in file charlie.bhv is shown in Listing 2 below. The first configuration block, in lines 3-5, are for parameters defined generally for all IvP Helm behaviors, and the second block, in lines 7-11, are for parameters defined explicitly for the Loiter behavior.

Listing 30.2: Configuration of the StationKeep behavior in the Charlie mission, from the file charlie.bhv.

```plaintext
0 //---------------------------------------------------------------
1 Behavior = BHV_StationKeep
2 {
3    name = station-keep
4    priority = 100
5    condition = MODE==STATION-KEEPING
6
7    center_activate = true
8    inner_radius = 5
9    outer_radius = 10
10   outer_speed = 1.0
11   swing_time = 0
12 }
```

30.4.1 Setting the Vehicle into the Station Keeping Mode

The vehicle is put into the station keeping mode by posting STATION_KEEP=true to the MOOSDB. This results in the helm mode, represented by the MOOS variable MODE, being set to station keeping. The hierarchical mode declarations used in the Charlie mission are declared in the top of the charlie.bhv file and were shown in Figure 71. The StationKeep behavior is conditioned on MODE==STATION-KEEPING, on line 5 above. The StationKeep behavior may be configured to station keep at a specified point in water (with the station_pt parameter), or it may be configured to station keep wherever it happens to be when it enters or re-enters the running state, as it is configured in the Charlie mission by virtue of line 7 above. In the Charlie mission, pMarineViewer is
configured with four on-screen buttons. For more info, see the "Command-and-Control" section of the pMarineViewer documentation, [5]. Two of the buttons are used for toggling the STATION\_KEEP variable to the MOOSDB.

### 30.4.2 What is Happening in the Station Keeping Mode

The station keeping behavior (the only running behavior in this mode) attempts to keep the vehicle within a certain distance, given by the inner_radius parameter, to a center point. Inside this radius, it simply lets the vehicle drift. Outside the radius it sets a heading and speed to drive the vehicle back to the center point. The speed decreases as it approaches the inner circle, and is at its highest speed (set by the outer_speed parameter on line 10 above) when its range to the center point is greater than that given by the outer_radius distance. It also makes two postings to the VIEW\_POLYGON object to represent the circles implied by the inner_radius and outer_radius parameters, as shown in Figure 75.

### 30.5 Suggestions for Tinkering in the Charlie Mission

- Station keeping is trivial if the vehicle doesn’t drift. Try turning on the artificial wind gusts available in the Charlie mission, courtesy of the uTimerScript script, available by selecting WIND\_GUSTS=true from the Action pull-down menu. The vehicle performance is a bit more interesting now in the station keeping mode.

- Try configuring the simulator, uSimMarine, to reflect a vehicle that takes much more time to come to a full stop in the water. This can be done by setting deceleration=0.1 in the uSimMarine configuration block, or by posting USM\_DECELERATION=0.1 to the MOOSDB once the mission is running. Note that when the vehicle is put into station keeping mode, its station point is set to the point where it is when it enters this mode. Since the vehicle takes so long to slow down, it immediately drifts out of the inner radius and turns around 180 degrees. This is a typical situation seen in the field. This can be countered a bit by setting the swing_time parameter in the StationKeep behavior. This parameter is the number of elapsed seconds after the vehicle enters the station keeping mode before the station keeping behavior marks its present position as the point to station keep around. Try setting swing_time=5 and re-running the above to see the difference.
Figure 72: The Charlie Mission (1): The vehicle "charlie" proceeds to a loiter polygon, traversing in a counter-clockwise manner to the first waypoint labelled "charlie's next waypoint".

Figure 73: The Charlie Mission (2): The loiter traversal polygon for the vehicle "charlie" has been dynamically changed to an ellipse via a selection from the ACTION pull-down menu.
Figure 74: **The Charlie Mission (3):** The vehicle, "charlie" recovers from a wind gust and proceeds to re-enter the polygon trajectory at the tangential vertex labelled "charlie's next waypoint".

Figure 75: **The Charlie Mission (4):** The vehicle "charlie" is put into the station keeping mode. The two rings around the vehicle are the inner_radius and outer_radius parameters.
31 The Delta Mission

The Delta example mission is used to illustrate the operation of a vehicle in the depth plane (an underwater as opposed to surface vehicle), the illustration of the PeriodicSurface behavior, the use of the Waypoint behavior with survey patterns, and the use of the pMarineViewer application to send field-control commands to alter the mission as it unfolds. The vehicle will initially deploy using the Loiter behavior, and will periodically come to the surface. If commanded by the user, the vehicle will break off from the Loiter behavior to execute a survey pattern, after which it will resume loitering at its original location.

31.1 Overview of the Delta Mission Components and Topics

<table>
<thead>
<tr>
<th>Behaviors:</th>
<th>Waypoint, Loiter, PeriodicSurface, ConstantDepth</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOOS Apps:</td>
<td>pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer, uTimerScript</td>
</tr>
<tr>
<td>Primary Topics:</td>
<td>(1) Configuring the Helm for operation at depth</td>
</tr>
<tr>
<td></td>
<td>(2) The ConstantDepth behavior</td>
</tr>
<tr>
<td></td>
<td>(3) The PeriodicSurface behavior</td>
</tr>
<tr>
<td></td>
<td>(4) The Waypoint behavior with survey patterns</td>
</tr>
<tr>
<td></td>
<td>(5) Using pMarineViewer with Geo-referenced mouse clicks</td>
</tr>
<tr>
<td>Side Topics:</td>
<td>(1) uTimerScript is used to simulate a UUV receiving a GPS update.</td>
</tr>
</tbody>
</table>

31.2 Launching the Delta Mission

The delta mission may be launched from the command line:

```
$ cd moos-ivp/missions/s4_delta/
$ ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 76, with a single vehicle, "dudley", initially in the PARK helm state. See Section 5.2 for more on the helm state. After hitting the DEPLOY button in the lower right corner, the vehicle enters the DRIVE state, and the LOITERING helm mode, and begins to proceed along the waypoints as shown. See Section 6.4 for more on more on the helm mode, vs. the helm state. The mode declarations for the Delta mission are defined at the top of the delta.moos file, and amount to the following simple hierarchy:

```
 o ROOT
   |-- o INACTIVE
   |-- o ACTIVE
     |-- o RETURNING
     |-- o SURVEYING
     |-- o LOITERING
```

In the initial helm mode after deployment, the LOITERING mode, the helm is running the Loitering behavior, Section 11, the ConstantDepth behavior, Section 15, and the PeriodicSurface behavior, Section 14. It will stay in this mode indefinitely until it is either commanded to return or break off to another region to conduct a survey pattern. In the LOITERING mode the vehicle will periodically
come to the surface for a GPS fix, presumably to correct for accumulated navigation error. The uTimerScript utility is used to simulate the event of receiving a GPS fix. This script is described in more detail in the uTimerScript documentation in the Section "A Script Used as Proxy for an On-Board GPS Unit".

In the SURVEYING mode, the Waypoint behavior becomes active and will execute a survey lawn-mower type pattern, as shown in Figures 77 and 79. In this mode, the periodic surfacing for GPS fixes is suppressed until the pattern is completed. The SURVEYING mode is entered whenever the MOOS variable SURVEY is set to true. In this mission, pMarineViewer is configured to toggle the SURVEY MOOS variable with on-screen buttons, and it is configured to accept mouse clicks which not only put the vehicle into the SURVEYING mode, but also accept the location of the mouse click as the center location of a predefined survey pattern. This kind of pMarineViewer configuration is discussed in Section 31.6.

31.3 Topic #1: Configuring the Helm for Operation at Depth

The Delta mission is configured to simulate a UUV and the helm is configured to produce decisions about the decision variables heading, speed, as well as depth. Adding depth to the helm decision space is done in the helm configuration block as shown below in Listing 1, taken from the delta.moos mission file. In this case, the depth decision space is defined on line 13 with a range from 0 to 500 meters, with a resolution of 1 meter. See Section 5.4.6 for more on configuring the helm decision space with the domain parameter.

Listing 31.1: Configuring pHelmIvP in the Delta mission to use behaviors concerning vehicle depth.

```plaintext
1 // pHelmIvP config block
2
3 ProcessConfig = pHelmIvP
4 {
5   AppTick = 4
6   CommsTick = 4
7
8   behaviors = delta.bhv
9   verbose = quiet
10  domain = course:0:359:360
11  domain = speed:0:4:21
12  domain = depth:0:500:501
13 }
```

In this example, the depth decision variable is a mandatory variable, along with heading and speed. This means that, on any given helm iteration, there must be a decision made about depth, or else the helm will post a helm error. How does the helm ensure that a depth decision is always part of any decision? The helm must be configured such that a behavior that reasons about depth is always active regardless of the helm mode. In the Delta mission, the three primary modes, LOITERING, SURVEYING, RETURNING, are all sub-modes of the ACTIVE mode, as shown above. The ConstantDepth behavior is configured to be running whenever the helm is in the Active mode.

31.4 Topic #2: The ConstantDepth Behavior

The ConstantDepth behavior is used in the Delta mission to keep the vehicle a prescribed depth while it is transiting, surveying, and loitering. For simplicity a single ConstantDepth behavior is
used, but a different behavior instance could also be used for each vehicle mode. The behavior configuration, in file delta.bhv is shown in Listing 2 below. The first part of the configuration block, in lines 3-6, are for parameters defined generally for all IvP Helm behaviors, and the second part, on line 8, is a parameter defined for the ConstantDepth behavior.

Listing 31.2: The ConstantDepth behavior in the Delta mission, from the file delta.bhv.

```
1 Behavior = BHV_ConstantDepth
2 {
3   name = bhv_const_depth
4   pwt = 100
5   duration = no-time-limit
6   condition = MODE==ACTIVE
7   depth = 15
8 }
```

The ConstantDepth behavior’s primary parameter is the `depth` parameter on line 8. This behavior does make provisions for providing a range of depths, or a more gradually degrading utility function when this behavior is used to work with other behaviors reasoning about depth. In this mission however, the depth is mostly non-contentious between behaviors. The exception is when the PeriodicSurface behavior is active, in which case the priority weight for the PeriodicSurface behavior is simply set to suppress the ConstantDepth behavior.

31.5 Topic #3: The PeriodicSurface Behavior

The PeriodicSurface behavior, described generally in Section 14, is used in the Delta mission to bring the vehicle to the surface for GPS fixes periodically, to simulate the need for occasional navigation corrections. The behavior configuration, in file delta.bhv is shown in Listing 3 below. The first part of the configuration block, in lines 3-6, is for parameters defined generally for all IvP Helm behaviors, and the second part, in lines 8-13, is for parameters defined for the PeriodicSurface behavior.

Listing 31.3: The PeriodicSurface behavior in the Delta mission, from the file delta.bhv.

```
1 Behavior = BHV_PeriodicSurface
2 {
3   name = bhv_periodic_surface
4   pwt = 1000
5   condition = (MODE == LOITERING) or (MODE == RETURNING)
6   condition = PSURFACE = true
7   period = 120
8   zero_speed_depth = 2
9   max_time_at_surface = 60
10  ascent_speed = 1.0
11  ascent_grade = fullspeed
12  mark_variable = GPS_UPDATE_RECEIVED
13 }
```

The PeriodicSurface behavior outputs an objective function solely on the `depth` decision variable. Note the priority weight in line 4 in Listing 3 is set to 1000, in comparison to the priority weight set for the ConstantDepth behavior on line 4 in Listing 2. When the PeriodicSurface behavior is running,
it may or may not be active and producing an objective function over the depth decision variable. When it is active, the influence on depth simply overrules the influence from the ConstantDepth behavior due to these relative priority weights. For this reason, the ConstantDepth behavior may be conditioned on all active modes as in line 6 of Listing 2.

Note the parameter setting on line 13 above, setting the "mark variable" to GPS_UPDATE_RECEIVED. This is the default value, and the line could be safely be removed from the configuration block, but it is included anyway to be clear. The PeriodicSurface behavior, once it has noted that the vehicle has reached the surface, will wait until a value has been posted to GPS_UPDATE_RECEIVED that is different than its previous posting. Usually a timestamp suffices. The configuration of uTimerScript to achieve simulated GPS updates is described in more detail in the uTimerScript documentation in the Section "A Script Used as Proxy for an On-Board GPS Unit".

### 31.6 Topic #4: The Waypoint Behavior with Survey Patterns

The Waypoint behavior, described generally in Section 9, is used in the Delta mission to conduct survey patterns like that shown in Figure 77. The behavior configuration, in file delta.bhv is shown in Listing 4 below. Note the survey pattern, described on line 15, does not list a set of points, but instead describes the pattern in terms of its properties such as the lane width and pattern orientation. This format is supported generally for building SegList objects from string specifications, discussed in the Geometry documentation.

Listing 31.4: The Waypoint behavior in the Delta mission, from the file delta.bhv.

```
1 Behavior = BHV_Waypoint
2 {
3   name = waypt_survey
4   pwt = 100
5   condition = MODE==SURVEYING
6   perpetual = true
7   updates = SURVEY_UPDATES
8   endflag = SURVEY = false
9
10   lead = 8
11   lead_damper = 1
12   speed = 2.0  // meters per second
13   radius = 8.0
14   points = format=lawnmower, label=dudley_survey, x=80, y=-80,
        width=70, height=30, lane_width=8, rows=north-south,
        degs=30
15 }
```

The waypoint behavior is configured to execute the survey pattern once, since the repeat parameter is not set and defaults to zero. Upon completion, it posts an end flag, configured on line 8, SURVEY=false to the MOOSDB. This immediately moves the vehicle out of the SURVEYING and into either the LOITERING or RETURNING mode depending on what it was doing when it was commanded to begin the survey. See the hierarchical mode declarations at the top of the delta.bhv file. Since the behavior is set with perpetual=true on line 6, completion of the survey pattern merely puts the behavior in a virtual stand-by mode until it is given a new set of waypoints and it once again meets its run conditions.
The waypoint behavior is configured to accept dynamic parameter updates on line 7 through the MOOS variable \textbf{SURVEY_UPDATES}. In the Delta mission, updates are initiated by a mouse click in the \texttt{pMarineViewer} window, with the result being a post to the MOOSDB of the \textbf{SURVEY_UPDATES} variable. Such a posting would consist of a new value for the \texttt{points} parameter, replacing the initial configuration on line 14. This is discussed next.

\subsection{31.7 Topic #4: Using \texttt{pMarineViewer} with Geo-referenced Mouse Clicks}

In the Delta mission, the \texttt{pMarineViewer} application is used to post messages to the MOOSDB to (a) put the vehicle into the \texttt{SURVEYING} mode from either the \texttt{LOITERING} or \texttt{RETURNING} modes and (b) grab a user specified point in the operation area, from the user’s mouse click, to be used as the center point of the commanded survey pattern. This situation is shown in Figure 77. This is achieved by utilizing the Mouse Context feature of \texttt{pMarineViewer}, discussed in the \texttt{pMarineViewer} documentation, \cite{5}, in the section "Contextual Mouse Poking with Embedded OpArea Information". The below two lines are inserted into the \texttt{pMarineViewer} configuration block in \texttt{delta.moos}:

\begin{verbatim}
left_context[survey-point] = SURVEY = true
left_context[survey-point] = SURVEY_UPDATES = points = vname=$(VNAME), \ 
x=$(XPOS), y=$(YPOS), format=lawnmower, label=delta, width=70, \ 
height=30, lane_width=8, rows=north-south, degs=80
\end{verbatim}

Both lines declare that a MOOS variable-value pair is to be posted whenever a left mouse click is generated by the user. The fist line sets \texttt{SURVEY=true}, in an attempt to put the helm into the \texttt{SURVEYING} mode. This alone will not suffice to switch modes if the current value of the MOOS variable \texttt{RETURN} is set to \texttt{true}. See the mode declarations near the top of the \texttt{delta.bhv} file for this mission - a command to return overrides a command to perform a survey. The second line above associates with a left-mouse click a posting to the \texttt{SURVEY_UPDATES} variable. Recall from Listing 4 that this is the very variable through which the surveying waypoint behavior is configured to receive dynamic parameter updates. This posting is configured with a few macros, \texttt{$(VNAME)$}, \texttt{$(XPOS)$}, \texttt{$(YPOS)$}, which are filled in with the name and position of the current vehicle in the \texttt{pMarineViewer} window. Each user left-mouse click thus re-assigns the vehicle to a new survey pattern, and switches the helm mode, unless it has been commanded to return.

\subsection{31.8 Suggestions for Further Experimenting with the Delta Mission}

\subsubsection{31.8.1 Failing to Reason about Depth}

In this mission, \texttt{depth} is a mandatory helm decision variable, along with \texttt{course} and \texttt{speed}, as configured in \texttt{delta.moos} and shown in Listing 1. If a mission is not configured properly it’s possible to bring about a helm mode where no behavior is reasoning about depth. Declaring \texttt{depth} to mandatory is equivalent to declaring that a situation where a helm iteration with no decision on \texttt{depth} is an error condition worth of an all-stop.

One way to bring this about in this mission, is to reconfigure the \texttt{ConstantDepth} behavior configuration in \texttt{delta.bhv} to replace line 6 in Listing 2 with the following

\begin{verbatim}
condition = (MODE==LOITERING)
\end{verbatim}
By doing this, there will be no depth related behavior when the vehicle is in the RETURNING mode. Try re-running the mission. Note when the mission is first launched, the label next to the vehicle in pMarineViewer reads "dudley (PARK)(ManualOverride)". This is normal and indicates that the helm state is PARK, simply because the operator retains manual control. In addition to seeing this in the pMarineViewer window, this can also be confirmed by scoping on a few key helm variables including IVPHELM_STATE and IVPHELM_ALLSTOP:

```
$ uXMS delta.moos IVPHELM_STATE IVPHELM_ALLSTOP BHV_ERROR --show=source,time,community
```

should produce something similar to:

<table>
<thead>
<tr>
<th>VarName</th>
<th>Source</th>
<th>Time</th>
<th>Community</th>
<th>VarValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVPHELM_STATE</td>
<td>pHelmIvP</td>
<td>81.86</td>
<td>dudley</td>
<td>&quot;PARK&quot;</td>
</tr>
<tr>
<td>IVPHELM_ALLSTOP</td>
<td>pHelmIvP</td>
<td>1.69</td>
<td>dudley</td>
<td>&quot;ManualOverride&quot;</td>
</tr>
<tr>
<td>MODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the IVPHELM_ALLSTOP value is posted once, until its value changes, and IVPHELM_STATE posts its value on every helm iteration regardless of the value. The latter variable also serves the purpose of a helm heartbeat indicator.

After noting the above, launch the mission by hitting the DEPLOY button, and note that the label next to the vehicle has changed to "dudley (LOITERING)". The four MOOS variables from above have also changed to:

<table>
<thead>
<tr>
<th>VarName</th>
<th>Source</th>
<th>Time</th>
<th>Community</th>
<th>VarValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVPHELM_STATE</td>
<td>pHelmIvP</td>
<td>109.56</td>
<td>dudley</td>
<td>&quot;DRIVE&quot;</td>
</tr>
<tr>
<td>IVPHELM_ALLSTOP</td>
<td>pHelmIvP</td>
<td>98.07</td>
<td>dudley</td>
<td>&quot;clear&quot;</td>
</tr>
<tr>
<td>MODE</td>
<td></td>
<td></td>
<td></td>
<td>&quot;ACTIVE:LOITERING&quot;</td>
</tr>
</tbody>
</table>

After the vehicle has been running a bit, try hitting the RETURN button which changes the helm to the RETURNING mode. In this case, due to our tinkering with the mission above, there is no behavior reasoning about depth, and the following message appears next to the vehicle on the screen: "dudley (Returning)(MissingDecVars:depth)". The four scoped MOOS variables also change to the following:

<table>
<thead>
<tr>
<th>VarName</th>
<th>Source</th>
<th>Time</th>
<th>Community</th>
<th>VarValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVPHELM_STATE</td>
<td>pHelmIvP</td>
<td>621.34</td>
<td>dudley</td>
<td>&quot;DRIVE&quot;</td>
</tr>
<tr>
<td>IVPHELM_ALLSTOP</td>
<td>pHelmIvP</td>
<td>617.09</td>
<td>dudley</td>
<td>&quot;MissingDecVars:depth&quot;</td>
</tr>
<tr>
<td>BHV_ERROR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODE</td>
<td></td>
<td></td>
<td></td>
<td>&quot;ACTIVE:RETURNING&quot;</td>
</tr>
</tbody>
</table>

247
Note that, despite the error and all-stop event, the helm remains in drive, and may return to loitering or surveying at any time. Try configuring the helm in `delta.moos` to contain the line `park_on_allstop=true` and note the difference in the above experiment. The generated error will bump the helm into the `PARK` helm status.
Figure 76: The Delta Mission (1): The vehicle "dudley" heads to its loiter region where it will remain indefinitely until it is commanded to return or perform a survey pattern. It will periodically surface for a GPS fix.

Figure 77: The Delta Mission (2): The user has clicked a point in the viewer around which a survey pattern is built. The vehicle exits the loitering mode and begins to execute the survey pattern.
Figure 78: **The Delta Mission (3):** Once the vehicle has finished the survey pattern, it re-enters the loiter mode returning to its loiter region. It resumes periodic surfacing and immediately surfaces for a GPS fix.

Figure 79: **The Delta Mission (4):** After resumption of loitering, the user clicks in the viewer a new point around which a new survey pattern is built. The entry point of the survey pattern automatically accommodates the vehicle.
32 The Echo Mission

The primary purpose of the Echo example mission is to illustrate the use of dynamically spawned behaviors. A simple behavior, the BearingLine behavior, is used to illustrate the idea. The BearingLine behavior simply posts a viewable point and viewable line segment representing the bearing from the present position of the vehicle to a fixed point in the operation area. Each new ”bearing point” posted to the MOOSDB results in a newly spawned BearingLine behavior in the helm.

32.1 Overview of the Echo Mission Components and Topics

<table>
<thead>
<tr>
<th>Behaviors:</th>
<th>Waypoint, BearingLine</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOOS Apps:</td>
<td>pHelmIvP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer</td>
</tr>
<tr>
<td>Primary Topics:</td>
<td>(1) The BearingLine behavior.</td>
</tr>
<tr>
<td></td>
<td>(2) Dynamic behavior spawning.</td>
</tr>
<tr>
<td></td>
<td>(3) Sending updates to the original and spawned behaviors.</td>
</tr>
<tr>
<td>Side Topics:</td>
<td>(1) uTimerScript is used to auto-generate events with random components that lead to behavior spawning.</td>
</tr>
<tr>
<td></td>
<td>(2) uHelmScope may be used to monitor the spawning and death of behaviors in the helm.</td>
</tr>
</tbody>
</table>

32.2 Launching the Echo Mission

The echo mission may be launched from the command line:

```
$ cd moos-ivp/missions/s5_echo/
$ ./launch.sh --warp=10
```

This should bring up a pMarineViewer window like that shown in Figure 80, with a single vehicle, henry, initially having the "PARK" helm state. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "SURVEYING" mode and begins to proceed along the waypoints as shown.

32.3 Topic #1: The BearingLine Behavior

In Figure 80, note the line segment rendered from the vehicle in the direction of point (100, -100). This line segment is posted to the MOOSDB by the IvP Helm on behalf of the BearingLine behavior. This behavior does not influence the trajectory of the vehicle at all. The line segment acts as an easy visual confirmation that the behavior is instantiated and running properly. In the Echo mission this behavior is configured as in Listing 1. The bearing line originates from the present vehicle position toward the point specified on line 8. The length of the line is 50% of the present distance between the vehicle and the bearing point, as specified on line 7. The bearing point is also configured to be rendered by the configuration on line 9.

*Listing 32.1: Configuration of the BearingLine behavior in the Echo example mission.*

```
1  Behavior = BHV_BearingLine
2  {
```
The BearingLine behavior produces no (IvP) objective function, so by the definition of the behavior run states (Section 6.5.3), it is never in the active state. In the Echo mission, the BearingLine behavior is in the running state when the vehicle is surveying the five waypoints shown in Figure 80. This can be confirmed by launching a uHelmScope window:

```
$ uHelmScope moos-ivp/ivp/missions/s5_echo/echo.moos -x -p}
```

The output should be similar to that shown in Listing 2 below. Note the Waypoint behavior is active, line 12, and the BearingLine behavior is in the running state, line 14.

Listing 32.2: Output of the uHelmScope tool during the execution of the Echo mission.

```
1 ============== uHelmScope Report ============== DRIVE (8)
2 Helm Iteration: 173   (hz=0.25)(5)   (hz=0.25)(100)   (hz=0.26)(max)
3 IvP functions: 1
4 Mode(s):
5 SolveTime: 0.00   (max=0.01)
6 CreateTime: 0.00   (max=0.01)
7 LoopTime: 0.00   (max=0.01)
8 Halted: false   (0 warnings)
9 Helm Decision: [speed,0,4,21] [course,0,359,360]
10 course = 189.0
11 speed = 2.0
12 Behaviors Active: ---------- (1)
13 waypt_survey (43.0) (pwt=100.00) (pcs=6) (cpu=0.17)
14 Behaviors Running: --------- (1)
15 bng-line (43.0) (upd=0/0)
16 Behaviors Idle: ----------- (1)
17 waypt_return
18 Behaviors Completed: ------- (0)
19 # MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
20 @ BEHAVIOR-POSTS TO MOOSDB ------------------------ (Hit '@' to en/disable)
```

### 32.4 Topic #2 Dynamic Behavior Spawning

The Echo mission is configured to allow dynamic behavior spawning for the BearingLine behavior. Lines 4 and 5 in Listing 1 allow the spawning by configuring templating to be enabled on line 4, and specifying the MOOS variable through which spawning requests are received on line 5. Recall that templating can be enabled with either the "clone" or "spawn" options. In this case, "clone"
option was chosen to allow the instantiation of one initial instance upon helm start-up, with the parameter configuration shown.

In this example, the pMarineViewer is configured to convert left-mouse clicks into posts of the BEARING_POINT variable to the MOOSDB, triggering the spawning of new BearingLine instances. A mouse click over the point (-5, -58) results in the post:

```
BEARING_POINT = "name=bng-line-5.0--158.0, bearing_point=-5.0,-158.0"
```

The helm receives mail for the BEARING_POINT variable since it registers automatically for each variable specified in any behavior configured with the updates parameter. The helm examines this string before applying the update, and notes that the behavior name specified is unique (not currently instantiated) and rather than interpreting this as a request to update the existing BearingLine behavior already instantiated, interprets it as a request to spawn a new BearingLine instance if templating is enabled (which it is). The new behavior is spawned, with the behavior name specified. In this case the behavior name is based on the coordinates of the point clicked by the user. Two successive clicks on the same point will result in two posts to BEARING_POINT by pMarineViewer, but the second post will be effectively ignored by the helm. (It is read by the helm, but since the behavior name is one that is already known to the helm, the update is applied to that existing behavior instance. In this case such an update to the bearing_point parameter would be redundant.

After two such user mouse clicks, there will be two new BearingLine behaviors instantiated, and the situation would look similar to that shown in Figure 81. If the uHelmScope tool is still connected as above, the output would look similar to that shown in Listing 3. Note the existence of three running BearingLine behavior instances reported in lines 15-17. The instance on line 15 was created upon helm startup, and the instances on lines 16-17 were created upon the user mouse clicks.

**Listing 32.3: Output of the uHelmScope tool during the later execution of the Echo mission.**

```
1 ============= uHelmScope Report ============= DRIVE (12)
2 Helm Iteration: 279  (hz=0.25)(5)  (hz=0.25)(100)  (hz=0.26)(max)
3 IvP functions:  1
4 Mode(s):     ACTIVE:SURVEYING
5 SolveTime:   0.00  (max=0.01)
6 CreateTime:  0.00  (max=0.01)
7 LoopTime:    0.00  (max=0.01)
8 Halted:     false  (0 warnings)
9 Helm Decision: [speed,0,4,21] [course,0,359,360]
10 course = 180.0
11 speed = 2.0
12 Behaviors Active: ---------- (1)
13 waypt_survey (69.6) (pwt=100.00) (pcs=4) (cpu=0.28)
14 Behaviors Running: ---------- (3)
15 bng-line (69.6) (upd=0/0)
16 bng-line-5--58 (66.1) (upd=1/1)
17 bng-line13--124 (56.8) (upd=1/1)
18 Behaviors Idle: ---------- (1)
19 waypt_return
```
32.5 Topic #3: Sending Updates to the Original and Spawned Behaviors

The action associated with the left-mouse click in the Echo mission is configured in the pMarineViewer configuration block in echo.moos by:

```
left_context[bng_point] = BEARING POINT = name=bng-line$(X)$-(Y) # bearing_point=$(X)$,$(Y)
```

Setting the context for the left and right mouse clicks in pMarineViewer is achieved by utilizing the Mouse Context feature of pMarineViewer, discussed in the pMarineViewer documentation, [5], in the section "Contextual Mouse Poking with Embedded OpArea Information". With the above configuration a left-mouse click may result in the following if clicked at the point (-5, -58):

```
BEARING POINT = name=bng-line-5--58 # bearing_point=-5,-58
```

Updating or changing the prevailing parameters for existing behaviors is possible via the use of the MOOS variable specified in the updates parameter for each behavior. The only difference between an update that changes parameters of an existing behavior, and an update that spawns a new behavior is the inclusion of a name=<behavior-name> as above. If this component is present in the string posted to the MOOSDB and the <behavior-name> specifies an existing behavior, then that behavior only will have its parameters updated. If it does not specify an existing behavior, a new behavior will be spawned with the specified name. If the name=<behavior-name> component is not included in the posting, then the update will be applied to all behaviors configured to receive updates via that particular MOOS variable.

This case is a bit interesting since all newly spawned behaviors specify the same MOOS variable for receiving updates. Indeed a single poke to the MOOS variable BEARING POINT could result in the simultaneous configuration modification of all instantiated BearingLine behaviors. In the Echo mission, pMarineViewer is configured to make the following pokes to the MOOSDB via the ACTION pull-down menu:

```
action = BEARING POINT = show_pt=true
action+ = BEARING POINT = show_pt=false
action = BEARING POINT = line_pct=0
action = BEARING POINT = line_pct=25
action = BEARING POINT = line_pct=50
action = BEARING POINT = line_pct=75
action+ = BEARING POINT = line_pct=100
action = BEARING POINT = "name=bng-line # line_pct=0"
action = BEARING POINT = "name=bng-line # line_pct=50"
action = BEARING POINT = "name=bng-line # line_pct=100"
```
The first two actions will turn on or off the rendering of the bearing points posted by each behavior. The next five actions will adjust the rendering length of the posted bearing line by each behavior. The last two actions will adjust the rendering length of the posted bearing line only for the behavior named "bng-line", the behavior spawned at the time of helm start-up.

### 32.6 Suggestions for Tinkering

- After the mission is launched, use the ACTION pull-down menu in pMarineViewer to select AUTO_SPAWN=true. This enables a script via uTimerScript that automatically generates new bearing points, spawning new behaviors. These behaviors have their durations set randomly by the script to be in range [5, 10] seconds. Note the new bearing lines emerging and moving with the vehicle until the behavior dies. The script will lead to the spawning (and death) of 5000 behaviors over the course of about two hours of simulation at MOOSTimeWarp=1.

- With AUTO_SPAWN=true as above, let the vehicle return to the launch point (by clicking the RETURN button in the pMarineViewer window. After it reaches the return point and perhaps sits for a bit, hit the DEPLOY button once again to return the vehicle into its SURVEYING mode. Notice that initially there are many bearing lines rendered before returning to only a handful of bearing lines after a few seconds of simulation. This is because newly spawned behaviors do not start their duration clock until the first time the behavior enters the running state. All behaviors spawned while returning to the start point are effectively put on hold until the vehicle is re-deployed. Then they all start their duration clocks simultaneously and all die off 5-10 seconds later.
Figure 80: **The Echo Mission (1):** The vehicle "henry" traverses waypoints with the Waypoint behavior. The BearingLine behavior generates a viewable "bearing point" at (100, -100), and a viewable "bearing line".

Figure 81: **The Echo Mission (2):** The user has clicked two new bearing points, (-5, -58), and (13, -124), and two new BearingLine behaviors have been spawned, each generating bearing line and bearing point visual outputs.
33 Mission S11: The Kilo Mission

The purpose of the Kilo mission is to illustrate the use of the standby helm, and the use of the TestFailure behavior. The standby helm is described in Section 5.7 and consists of an otherwise normally configured helm with the additional standby parameter invoked. This helm will wait in standby mode until the heartbeat of another (primary) helm ceases to be posted to the MOOSDB for some period of time. In this example mission, a primary and standby helm are configured with the primary helm executing a simple mission similar to the Alpha mission, but also using an instance of the TestFailure behavior. This behavior will be configured to trigger either a crash or hang of the primary helm to demonstrate the manner in which a standby helm will step in to take over with its own mission. The standby mission in this case is simply a behavior to return the vehicle to its launch position.

33.1 Overview of the Kilo Mission Components and Topics

<table>
<thead>
<tr>
<th>Behaviors:</th>
<th>Waypoint, TestFailure</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOOS Apps:</td>
<td>pHelmIVP, pLogger, uSimMarine, pMarinePID, pNodeReporter, pMarineViewer</td>
</tr>
</tbody>
</table>
| Primary Topics: | (1) The use of a standby helm.  
(2) The TestFailure behavior.  
(3) Scoping the primary and standby helm states at runtime. |
| Side Topics: | (1) uHelmScope may be used to monitor the relationship between the shadow helm and primary helm. |

33.2 Launching the Kilo Mission

The Kilo mission may be launched from the command line:

```
$ cd moos-ivp/missions/s11_kilo/
$ ./launch.sh --warp=10 (or more simply, ./launch.sh 10)
```

This should bring up a pMarineViewer window like that shown in Figure 85, with a single vehicle, "kilo", initially in the "PARK" mode. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "SURVEYING" mode and begins to proceed along the waypoints as shown.

33.3 Topic #1: The Use of a Standby Helm

The vehicle in this example is configured with both a primary helm and a secondary helm. Configuring a mission with two helm instances is straightforward. The primary helm is configured as done normally, as in lines 3-11 in Listing 1 below. The standby helm is configure identically but with one additional line, as in line 23 in Listing 1 below. This line denotes the helm it to be a standby helm, and indicates the time threshold used in determining when to take over from a primary helm.

```
Listing 33.1: Configuration of the primary and secondary helm in the Kilo mission.

1 //----------------------------- pHelmIVP config block
2
```
3 ProcessConfig = pHelmIvP
4 {
5    AppTick = 4
6    CommsTick = 4
7
8    behaviors = kilo.bhv
9    domain = course:0:359:360
10   domain = speed:0:4:21
11 }
12
13 //------------------------------------------ pHelmIvP_Standby config block
14
15 ProcessConfig = pHelmIvP_Standby
16 {
17    AppTick = 4
18    CommsTick = 4
19
20    behaviors = kilo_standby.bhv
21    domain = course:0:359:360
22    domain = speed:0:4:21
23    standby = 2
24 }

Both helms are instances of the pHelmIvP application, but one of them is named pHelmIvP_Standby. The name of the application is not in any way used to put this helm instance into a standby role. It is chosen simply to be different from the primary helm, since the MOOSDB requires all connected applications to have unique names, and to be clear when or if debugging the mission.

The shadow helm is launched by pAntler with the name pHelmIvP_Standby. This is done by giving it the alternative name using the pAntler ExtraProcessParams parameter as shown on lines 15 and 18 below.

Listing 33.2: Configuration of Antler.

1 //------------------------------------------
2 // Antler configuration block
3
4 ProcessConfig = ANTLER
5 {
6    MSBetweenLaunches = 200
7
8    Run = MOOSDB @ NewConsole = false
9    Run = pLogger @ NewConsole = false
10   Run = uSimMarine @ NewConsole = false
11   Run = pNodeReporter @ NewConsole = false
12   Run = pMarinePID @ NewConsole = false
13   Run = pMarineViewer @ NewConsole = false
14   Run = pHelmIvP @ NewConsole = true
15   Run = pHelmIvP @ NewConsole = true, ExtraProcessParams=HParams
16   Run = uProcessWatch @ NewConsole = false
17
18   HParams=--alias=pHelmIvP_Standby
19 }
If the shadow helm is launched independently, not with \texttt{p\Antler}, it may be launched with:

\begin{verbatim}
$ pHelmIvP kilo.moos --alias=pHelmIvP_Standby
\end{verbatim}

\section{33.4 Topic #2: The TestFailure Behavior}

The Kilo mission uses the TestFailure behavior described in Section 22 to artificially generate a failure of the primary helm. In the Kilo mission it is configured to produce a "hung" helm with the following configuration:

\begin{verbatim}
Listing 33.3: Configuration of the TestFailure Behavior.
1  //----------------------------------------------
2  Behavior = BHV_TestFailure
3  {
4      name = test_failure
5      condition = DEPLOY=true
6      duration = 120
7      duration_idle_decay = false
8
9      failure_type = hang,3
10  }
\end{verbatim}

The first four configuration parameters are defined for all IvP behaviors. A run condition of \texttt{DEPLOY=true} keeps this behavior idle until the mission is launched. Once it is launched, a countdown of 120 seconds begins until the behavior will hang. The \texttt{duration_idle_decay=false} setting ensures that the countdown doesn’t begin until the behavior is in the running state. In this case the behavior will fail by hanging for three seconds. This will cause the primary helm to also appear to hang for three seconds, noted by a three second gap between heartbeats, defined by posting to the MOOS variable \texttt{IVPHELM\_STATE}. Since the standby helm is configured to wait no longer than two seconds (see Listing 1), the standby helm will promptly take over control from the primary helm. The sequence of events in taking over control is described in more detail next.

\section{33.5 Topic #3: Scoping the Helm State(s) at Runtime}

The relationship between the primary and standby helm may be monitored by scoping on the variable \texttt{IVPHELM\_STATE} during the course of the mission. Figure 82 below shows the situation shortly after the vehicle is deployed using the \texttt{uXMS} scoping tool.
Initially the standby helm is showing a helm state of "STANDBY", and the primary helm is showing a helm state of "DRIVE". Since both helms are operating at the same frequency, they mostly alternate between postings as shown above.

The events shown in Figure 83 show that during the 9 identical postings ending at time 329.86, the standby helm is the only helm emitting a heartbeat. During this period the primary helm has either crashed or hung. Finally the secondary helm takes over and posts a helm state of "DRIVE+" for 107 consecutive heartbeats (iterations). Recall the "+" is to further distinguish that this helm was originally configured as a standby helm.

Eventually it turns out that the original primary helm did not crash after all but was just temporarily hung. After emerging from its hung state, upon reading mail in the next loop, it realizes the standby helm has taken over and immediately concedes control and posts the "DISABLED" helm state, and will never again post. One may wonder why the primary helm, when it finally woke up, did not first mistakenly post "DRIVE" before conceding control - after all it needs to first read its mail to learn that another helm is in control. The primary helm in fact did post "DRIVE" and "DISABLED" on two consecutive iterations. The first posting occurred (time 327.81) prior to querying the behaviors for input, and the second posting (time 356.70) occurred on the next iteration of upon reading mail.

In the case where the standby helm takes over for a crashed helm, the sequence of events, viewed...
from the perspective of posts to \texttt{IVPHELM\_STATE}, is similar as shown in Figure 84. When things are going fine the standby helm and primary helm alternately post helm states of \texttt{STANDBY} and \texttt{DRIVE} respectively as the first several posts below shoe. At some point the primary helm crashes and the only posts are made by the standby helm. In the example here, there are 10 \texttt{IVPHELM\_STATE="STANDBY"} posts made while the standby helm is getting closer to triggering a take-over. Finally, upon takeover, it posts a helm state of "\texttt{DRIVE+}\" thereafter. The primary helm is never heard from again.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure84.png}
\caption{Helm State in Kilo Mission after Standby Helm Takes Over Crashed Primary Helm: The standby helm has detected a delay in the primary helm heartbeat and takes over about two seconds later. Note the number in parentheses is the number of identical postings with the timestamp showing the time of the last of the identical postings.}
\end{figure}

\subsection*{33.6 Suggestions for Tinkering}

- \textit{Analyze a helm takeover from the log files.} Run the Kilo mission again and, as opposed to monitoring things with a live scope as in Figures 82 - 84, let the mission play out. Change directories to the folder where the \texttt{.alog} file was created. Take a look at the \texttt{IVPHELM\_STATE} postings using the \texttt{aloggrep} tool. Confirm that the sequence of events described in Figures 82 - 84 are consistent with entries of the log file. Hint: This can be achieved without mission configuration or code modifications.

- \textit{Crashing with helm with a mouse click.} Rather than waiting for the \texttt{TestFailure} behavior to run down its clock to failure, configure it to fail immediately upon entering the run state (\texttt{duration=0}). But also set an additional run-condition, e.g., \texttt{FAIL=true} and configure the \texttt{pMarineViewer} to make this posting upon a mouse-click. Test the modified Kilo mission by launching it, and clicking the mouse when ready to see the failure. This can be achieved solely through mission configuration modification, without code modification.

- \textit{Configure a standby helm that finishes a task.} Create a primary and shadow helm where the primary helm is surveying a set of waypoints. The shadow helm should be able to survey the remaining points when/if the primary helm is taken over. Hint: this will likely require code generation or modification. Extra bonus if achievable solely through mission modification.
Figure 85: **The Kilo Mission (1):** The vehicle "kilo" traverses the waypoints using a primary helm, launched alongside a standby helm. The TestFailure behavior is a ticking bomb that will hang the primary helm momentarily.

Figure 86: **The Kilo Mission (2):** The TestFailure behavior has hung the primary helm. The standby helm has detected the absence of a heartbeat (the IVPHELM\_STATE variable), and take over with a simple return mission.
34 The Berta Mission

The Berta mission involves two vehicles repeatedly performing collision avoidance with one another. The primary capabilities highlighted are the helm’s collision avoidance behavior, the basic contact manager, `pBasicContactMgr`, and the dynamic spawning and removal of behaviors related to contacts from the helm.

34.1 Overview of the Berta Mission Components and Topics

Here is a bit of what the Berta mission should look like:

![Figure 87: The Berta mission.](https://vimeo.com/84492574)

<table>
<thead>
<tr>
<th>Behaviors:</th>
<th>Loiter, AvoidCollision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicles:</td>
<td>Henry and Gilda</td>
</tr>
<tr>
<td>MOOS Apps:</td>
<td>pHelmIVP, pLogger, uSimMarine, pBasicContactMgr, pMarinePID, pNodeReporter, pMarineViewer, uTimerScript, pShare</td>
</tr>
</tbody>
</table>
| Primary Topics: | (1) The IvP Helm AvoidCollision behavior / Dynamic behavior spawning.  
(2) Contact management with the pBasicContactMgr application. |
| Side Topics: | (1) uTimerScript is used in a command and control role for random permuting of vehicle loiter assignments.  
(2) pShare is used for connecting one shoreside command and control community to two simulated vehicle communities.  
(3) The Loiter behavior is configured to receive dynamic updates on its loiter position. |
34.2 Launching the Berta Mission

The *Berta* mission may be launched from the command line:

```
$ cd moos-ivp/missions/m2_berta/
$ ./launch.sh --warp=10
```

This should bring up a *pMarineViewer* window like that shown in Figure 88, with two vehicles, *henry* and *gilda*, each initially in the "PARK" state. After hitting the DEPLOY button in the lower right corner, the vehicle enters the "LOITERING" mode and begins to proceed along the loiter pattern as shown. The example mission is configured to periodically alter the position of each vehicle’s loiter pattern, alternating between a random point in Region-A and Region-B shown in Figure 88. The user may click the PERMUTE-NOW button to immediately cause a new permutation and command sent to the vehicle, which otherwise happens automatically every three minutes.

34.3 Topic #1: The AvoidCollision Behavior and Dynamic Behavior Spawning

The AvoidCollision behavior runs on both vehicles and is configured identically on each vehicle in this mission. Its configuration is shown in Listing 1. When the helm first launches on each vehicle this behavior is not present. It is configured to be a dynamically spawned behavior - one for each known contact within a certain range of the vehicle. In this regard it depends on alerts from a contact manager running separately on each vehicle that posts the MOOS variable `CONTACT_INFO`, with the name of the new contact for which to spawn a new behavior.

The `CONTACT_INFO` variable is the variable for which the behavior is configured to receive dynamic updates, on line 6. The behavior is configured to allow dynamic instantiations on line 8 with `templating = spawn`. The choice of `spawn` (vs. `clone`) means that a behavior of this configuration is not initially present unless cued via the `updates` interface. See the discussion on dynamic behavior spawning in Section 7.7.

*Listing 34.1: Configuration of the AvoidCollision behavior in the Berta example mission.*

```
1 Behavior = BHV_AvoidCollision
2 {
3   name = avd_collision
4   pwt = 200
5   condition = AVOID=true
6   updates = CONTACT_INFO
7   endflag = CONTACT_RESOLVED = [CONTACT]
8   templating = spawn
9
10  contact = to-be-set
11  on_no_contact_ok = true
12  extrapolate = true
13  decay = 30,60
14
15  pwt_outer_dist = 50
16  pwt_inner_dist = 20
17  pwt_grade = linear
18  completed_dist = 75
19  min_util_cpa_dist = 8
```
The parameters in lines 15-17 determine the level of activity of the behavior based on the range to the contact. The parameters in lines 15 and 16 refer to the priority weight of the behavior. Beyond 50 meters in range, the weight will be 0% of its static weight of 200 assigned on line 4. (A weight of zero means that no objective function will be produced and the behavior is in the running state, but not the active state.) At 20 meters range, it will be at 100% of its static priority weight. The grade in priority between outer and inner distances is linear, due to the setting on line 17. The $\text{min}_\text{util}_\text{cpa}_\text{dist}$, and $\text{max}_\text{util}_\text{cpa}_\text{dist}$ on lines 19 and 20 refer to the utilities assigned to candidate maneuvers in the objective function output by the behavior. Any candidate maneuver whose CPA is less than 8 meters will be given the lowest utility rating, equivalent to an actual collision. Likewise, any candidate maneuver whose CPA is greater than 25 will be given the highest utility rating, equivalent to missing the contact by infinite distance.

The completed_dist, on line 18, refers to the range to the contact beyond which the behavior will declare itself to be complete. Once completed, the behavior will post its end flags (none in this case), and will remove itself from the helm. In this regard, some coordination with the contact manager is needed. At the very least, the contact manager must be configured to post alerts to the mutually configured MOOS variable, CONTACT_INFO, on line 6 in Listing 1 and on line 16 in Listing 2. Note also that the completed_dist is set to 75 meters, which is higher than the 55 meters set for the alert_range in the contact manager. This means that, after the avoid collision behavior dies, the contact must come back within 55 meters before the contact manager generates another alert, triggering the helm to once again spawn a behavior for the contact.

Once a contact has gone out of range to the point where the AvoidCollision behavior dies, it typically never returns, in practice. There is no guarantee of this however, since the future maneuvers of either the contact or ownership may once again put them on a collision course. The Berta mission tests this scenario repeatedly. The two vehicles are configured to loiter in a loiter pattern that keeps them out range from each other in terms of triggering an AvoidCollision behavior in each of their helms (as in Figure 88). In this mission, a uTimerScript script is running in the shoreside community that periodically picks a new loiter location within one of the two regions shown in Figure 88. It picks a new location once every three minutes, alternating between the two regions for each vehicle each time. This ensures that there will be a reasonably unpredictable collision avoidance situation each time the vehicles transit to their new loiter locations. The permutations happen automatically once every three minutes, but the user may jump to the next permutation by clicking on the PERMUTE-NOW button.

In Figure 89, note the line segment rendered between the two vehicles. This line segment is posted to the MOOSDB by the IvP Helm on behalf of the AvoidCollision behavior. A white line is drawn between the two vehicles as soon as the AvoidCollision behavior is spawned. The line turns a non-white color as soon as the behavior begins to generate an objective function, thus actively influencing the helm to prefer maneuvers with a lower expected CPA. The line is initially green to indicate a lower behavior priority. As the priority grows, the line turns yellow and finally red when it is at the higher priorities. The behavior posting of these line segments is optional and configurable. The configuration on line 21 in Listing 1 is responsible for the postings in this mission.
In this example, the `extrapolate` option is turned on for the AvoidCollision behavior (line 12 in Listing 1). Extrapolation is typically enabled when long durations are experienced between updates on contact position (which is not the case here in simulation). The behavior will automatically use the linear extrapolated position given the contact’s last known position, trajectory and time-stamp. The linear extrapolation will decay down to a zero speed (stationary solution) beginning at 30 seconds and coming to a complete stop at 60 seconds, due to the decay specification on line 13 of Listing 1.

### 34.4 Topic #2: Contact Management with pBasicContactMgr

In the Berta mission, the `pBasicContactMgr` is running on both simulated vehicles, configured identically with the configuration block shown in Listing 2. Both the contact manager and the helm are receiving `NODE_REPORT` messages containing information about contacts. The helm stores this information in its information buffer for any behavior that requests it during normal operation. The contact manager uses contact information to generate alerts, which potentially cues the helm to spawn new contact-related behaviors.

The contact manager may be configured to generate more than one alert. However, in the Berta mission, a single alert, on line 15 in Listing 2 is sufficient for spawning collision avoidance behaviors. Note the alert specifies a variable name, and variable value. The variable name, (`CONTACT_INFO`), is set to be the variable used for updates in the AvoidCollision behavior, on line 6 in Listing 1. The variable value is configured to be `#`-separated list of parameter-value pairs suitable as input to any behavior receiving updates. (See Section 7.2.5 on behavior updates.) In this case, the alert is configured to ”update” the AvoidCollision behavior parameters `name`, and `contact`. Both parameter values are comprised, in part, of the name of the contact, `$\{VNAME\}`. In the case of the behavior name, this is to ensure that the behavior name is unique, a condition for spawning a new behavior. It also ensures that multiple AvoidCollision behaviors will not be spawned for the same contact. In the case of the contact name, the behavior simply must know the name of the contact for which it avoiding a collision.

In the unlikely event that two distinct contacts are present in the field, and reporting the same vehicle name, it would be up to the contact manager to discern that these are indeed two contacts and not one. Currently this is beyond the algorithms implemented in `pBasicContactMgr`.

**Listing 34.2: Configuration of the pBasicContactMgr application in the Berta example mission.**

```plaintext
1 // pBasicContactMgr MOOS Configuration Block
2
3 ProcessConfig = pBasicContactMgr
4 {
5   AppTick     = 2
6   CommsTick   = 2
7
8   alert_range = 55
9   alert_cpa_range = 70
10  alert_cpa_time   = 30
11  contact_max_age = 3600
12  display_raddii = false
13  verbose       = true
14```
In the Berta example mission, the contact manager is configured to generate alerts when a known contact comes within 55 meters of the vehicle (line 8 of Listing 2). The contact manager also is configured to generate alerts when a contact comes within 70 meters (line 9) and its calculated CPA over next 30 seconds (line 10) falls within the alert range (again line 8). See the pBasicContactMgr documentation, [3], for more on the relationship between these two ranges.

The contact manager is capable of optionally posting to the MOOSDB two circles rendering the two ranges, the alert_range and alert_cpa_range. The rendering is turned on and off with the setting on line 12. The circles are posted under the variable VIEW_POLYGON in the local MOOSDBs on each vehicle, and each vehicle is configured to bridge this variable (via pShare) to the shoreside community running the pMarineViewer.

34.5 Suggestions for Tinkering in the Berta Mission

- Alter the alert ranges in the contact manager, and enable the displaying of range radii (line 8 in Listing 2) and notice when the AvoidCollision behavior is spawned, by the presence of the bearing lines between the two contacts.

- Alter the pwt_*_dist parameters in the AvoidCollision behavior (lines 15-16 in Listing 1) and check how this affects the ranges between the vehicles as they avoid collisions.
Figure 88: The Berta Mission (1): The two vehicles, *henry* and *gilda*, initially transit to their respective loiter stations and await further random location changes between the shown boxes.

Figure 89: The Berta Mission (2): The vehicles have changed their loiter stations and this puts them at risk for collision. The green bearing line between the vehicle indicates the presence of the AvoidCollision behavior.
35 Extending MOOS-IvP By Example

35.1 Brief Overview

This section describes an example repository distributed with the MOOS-IvP software bundle at www.moos-ivp.org. This repository merely provides a template with an example MOOS application, IvP Behavior, and example mission. More importantly perhaps is that the CMake build files are provided. A cursory look at these files reveals the hooks to add a new behavior or application. This is meant to provide one easy way to begin extending the MOOS-IvP software capabilities with one’s own modules.

35.2 Obtaining and Building the Example Extensions Folder

The example extensions folder is available at the following URL:

http://www.moos-ivp.org/software/extensions.html

Instructions are provided for downloading the software from an SVN server with anonymous read-only access. After checking out the tree from the SVN server as prescribed at this link, the top level directory should have the following structure:

```
$ cd moos-ivp-extend
$ ls
moos-ivp-extend/
  bin/
  docs/
  missions/
  src/
```

The build instructions are maintained in the README files and are probably more up to date than this document. In short building the software amounts to two steps:

```
$ cd moos-ivp-extend/src/
$ cmake ./
$ make
```

The build depends on the directory `moos-ivp-extend` being in the same directory as `moos-ivp`. If this needs to be different on your system, the file `CMakeLists.txt` in the `src/` directory can be edited. The relevant lines are at the top of the file:

```
GET_FILENAME_COMPONENT(MOOS_BASE_DIR_A ../../moos-ivp/trunk/MOOS ABSOLUTE)
GET_FILENAME_COMPONENT(IVP_BASE_DIR_A ../../moos-ivp/trunk/ivp ABSOLUTE)
GET_FILENAME_COMPONENT(MOOS_BASE_DIR_B ../../moos-ivp/MOOS ABSOLUTE)
GET_FILENAME_COMPONENT(IVP_BASE_DIR_B ../../moos-ivp/ivp ABSOLUTE)
```

After building the software there should be a new MOOS application called `pXRelayTest` in the `bin/` directory, and a new IvP Behavior in the directory `src/lib_behaviors-test/` directory. The new behavior is in the form of a shared object, having the name `libBHV_SimpleWaypoint.so` in Linux, and `libBHV_SimpleWaypoint.dylib` on the Mac OS X platform.
35.3 Using the New MOOS Application

To use the new MOOS application, the directory `moos-ivp-extend/bin/` needs to be added to the user’s shell path. This is typically done in the `.cshrc` or `.bashrc` file for tcsh and bash users respectively. To confirm that things are ready to go, use the built-in shell command `which`:

```
$ which pXRelayTest
```

which returns the directory where the executable resides if it is indeed in the shell’s path. Otherwise it returns nothing. Don’t forget that an edited path doesn’t take effect until a new shell is launched or unless the user types "source .cshrc", or "source .bashrc".

The `pXRelayTest` application is the same as the `pXRelay` application distributed with the MOOS-IvP software bundle. It differs only in name for the sake of illustrating the process of building a new application outside the `moos-ivp` tree. This example MOOS application is described in detail in Section 3.8. In that section, an example mission file is described for running two `pXRelay` processes to illustrate their function. A similar mission file is provided in:

```
$ moos-ivp-extend/missions/xrelay/xrelay.moos
```

that launches two processes, `pXRelay` and `pXRelayTest` as a way of confirming that you are running a MOOS application from the extensions build alongside the build of the main `moos-ivp` repository. Information on how to work through this example is provided in Sections 3.8.2 and 3.8.3.

35.4 Using the New IvP Helm Behavior

To use the new IvP Helm behavior built in the extensions folder, the helm needs to know about it. The helm already contains a number of behaviors compiled into the `pHelmIvP` executable, but the objective of adding behaviors in the manner outlined here, is to avoid any recompiling of the helm as new behaviors are added. Loosely speaking, there is a one-way dependency between repositories - new behaviors are layered onto the set of behaviors shipped with the helm with no modifications or re-build required of the basic `moos-ivp` software tree.

Newly built behaviors are compiled into shared object files, `*.so` in Linux, and `*.dylib` in Mac OS X. The helm references a path variable called `IVP_BEHAVIOR_DIRS` which contains a colon-separated list of all directories containing dynamically loadable behaviors. This variable is a shell environment variable and is typically set in the `.cshrc` or `.bashrc` file for tcsh and bash users respectively. For example, the following lines in the `.bashrc` file for bash users:

```
export IVP_BEHAVIOR_DIRS=/home/bob/moos-ivp-extend/src/lib_behaviors-test
```

A mission file to test this is provided in:

```
moos-ivp-extend/missions/alder/alder.moos
```

The mission is launched with:
The output produced in the helm terminal window should look like that shown in Listing 1 below, and provides useful feedback on whether the dynamically loadable behavior was loaded properly.

Listing 35.1: Example pHelmIvP terminal output when loading a dynamic behavior.

```bash
$ cd moos-ivp-extend/missions/alder/
$ pAntler alder.moos
```

The output prior to line 15 is standard MOOS output for an application connecting to the MOOSDB server. The lines thereafter are specific to the pHelmIvP application. In lines 16-19, the helm indicates that the directories specified in the IVP_BEHAVIOR_DIRS environment variable were found and indicates all dynamic behaviors loaded from those directories, regardless of whether they are used in this mission. Line 20 indicates the number of behavior files (.bhv files) comprising this mission. For each behavior file, output similar to lines 21-26 are generated which reports on the attempts to load individual behavior, noting for each whether they are a static behavior of a dynamically loaded behavior.

When the example is fully launched, the pMarineViewer should appear with a simulated vehicle, and two buttons at the lower right corner. The vehicle can be launched by clicking the "DEPLOY" button. The dynamically loaded behavior is called BHV_SimpleWaypoint and is described in detail in Section 40.
35.5 Extending the Extensions

To add further MOOS application modules, the simplest way by this example is to create sibling directories to the `pXRelayTest`, and add the corresponding entry to the `CMakeLists.txt` file in the `src/` directory. Further IvP behaviors can be added within the `lib_behaviors-test` directory, or in a separate `lib_*` directory. In the former case, the `CMakeLists.txt` file in the behavior directory needs to be augmented for the new behavior. In the latter case, an extra entry in the `CMakeLists.txt` file in the `src/` directory is required, as well as the addition of another directory in the `IVP_BEHAVIOR_DIRS` variable as described above in Section 35.4.
36 Introduction to the IvPBuild Toolbox

The IvPBuild Toolbox is a set of C++ classes and algorithms for building IvP functions. The primary objective is to provide tools to the implementors of new helm behaviors that are fast and easy to use. In the behavior implementation example in Listing 4, the creation of an IvP function required only about a dozen lines of code using two different methods available in the IvPBuild Toolbox.

36.1 Brief Overview

An instance of the class IvPFunction is the primary output of a behavior on each helm iteration, and is comprised of anywhere from a handful to thousands of "pieces" that approximate the utility function of the behavior. An example IvP function approximating a utility function is shown below in Figure 90.

Figure 90: An IvP function approximating an underlying function: The fview tool is used to render an IvP function with 289 pieces to approximate a given function, shown below the IvP function.

The toolbox contains tools for making simple one-variable objective functions (the "ZAIC" tools) as well as functions over N variables (the "Reflector" tools). The primary contribution of the user (behavior implementor) is to provide the underlying utility function provided to the toolbox. The IvP function approximation is generated automatically given user parameter preferences.
36.1.1 Where to Get the IvPBuild Toolbox

The IvPBuild Toolbox is part of the standard moos-ivp bundle distributed from www.moos-ivp.org. See Section ???. In the software tree it is entirely contained in the module `lib_ivpbuild`.

36.1.2 What is an Objective Function?

An objective function is a function like any other, a mapping from a domain to a range. In the case where the domain variables correspond to decisions or choices, and the range corresponds to the utility with respect to a particular user objective, the function is often called an ”objective” function, or ”utility” function.

36.1.3 What is Multi-objective Optimization?

The term multi-objective optimization refers to a situation where there are multiple objective functions defined over the same domain, i.e., decision space, and the ideal goal is to find a point in the decision space that optimizes all functions simultaneously. Rarely is such a mutually agreeable decision available and typically the functions can be said to be ”competing”. Techniques vary widely on how to handle this. A simple technique would be to rank order the functions and optimize the most important first, and so on. Another technique involves setting a competence threshold for each and choosing from decisions that satisfy a minimum competence for each function. For an in-depth treatment see [7], [42], [45], [49], [50], [51], [52], [53].

Many techniques for optimization are predicated on there being a user involved in the decision process who can interactively alter parameters of the problem until an agreeable resolution emerges. In these cases the notion of Pareto optimality, [48], often plays a central role. A Pareto optimal solution is one that cannot be improved in regard to one objective unless it comes at the expense of another objective. Typical user-interactive multi-objective optimization techniques involve letting the user explore the Pareto frontier, i.e., those solutions that are all Pareto optimal differing only on the user’s value function or relative preference in importance of objectives.

In repeatedly applying multi-objective optimization to the output of behaviors in an on-board autonomous decision making system, there is no user involved by definition. There is no exploration of the Pareto frontier since that exploration requires a user. Instead, part of the autonomy process involves also setting the value function, i.e., the relative importance of objectives. In the IvP helm, this value function is reflected by priority weights assigned to each function, and the multi-objective optimization problem is reduced to a single objective optimization problem, given $k$ functions and $w_i$ being the weight of the $i$th function:

$$
\hat{x}^* = \arg\max_{\hat{x}} \sum_{i=0}^{k-1} w_i f_i(\hat{x})
$$

The properties of IvP multi-objective optimization and solution algorithms were discussed in Section 6.6.3.
36.1.4 What is an IvP Function?

An IvP function is a piecewise linearly defined function where each piece has an upper and lower boundary (or interval) on the decision space and linear function defined over the piece. An IvP function is defined over a domain that itself has an upper and lower boundary for each decision variable. Furthermore, the domain is comprised of equally spaced discrete points, and therefore each piece is defined over a finite set of points in the domain. An IvP function is typically an approximation of the user’s underlying utility function. The fidelity of this approximation can be controlled by the user of the toolbox by deciding how many pieces are used in the approximation. Since the size or extents of each piece may vary within a function, the toolbox methods may also create functions that user smaller pieces where the underlying function is less amenable to local linear approximations.

An IvP function as an instance of the IvPFunction class defined as part of the lib_ivpcore module included in the basic software bundle distributed from www.moos-ivp.org.

36.1.5 Why the IvP Function Construct? A Brief Description of the Solver

The IvP function construct was chosen because it balances three aspects needed for use in the extendable behavior-based autonomy philosophy.

- **Flexibility**: a piecewise defined function approximation can be formed from any underlying function and thus the behavior author is not compelled to produce objective functions of a restricted form, such as convex or continuous functions. The behavior author is free to innovate. The behavior author typically has insight into the degree of fidelity needed to faithfully reflect the underlying utility function.

- **Speed**: the IvP function constructs, once produced, can be exploited by solution algorithms to give very fast solutions with a guarantee of global optimality modulo the error introduced in function approximation.

- **Accuracy**: a piecewise defined function can be highly accurate for a few reasons, (a) by controlling the piece size and distribution the approximation can be made to be as accurate as needed. (b) by being free to approximate any underlying function form, a piecewise function may better reflect a behavior’s utility. (c) by allowing for guaranteed globally optimal solutions in the resulting optimization problem, errors of this type are eliminated.

The IvP Solver uses a branch-and-bound method to search through the combination space of pieces, one from each of k contributing function. Since each point in the decision space is contained in exactly one piece in each function, the optimal decision corresponds to a k-tuple of pieces. Thus finding the optimal k-tuple guarantees that the optimal point in the decision space has been found. A leaf node in the tree is simply the ”intersection” of pieces from contributing functions. Likewise the intersection of interior functions at a leaf node is simply the sum of the linear functions of each contributing piece. Both the intersection of rectilinear pieces and the sum of linear functions be rapidly and simply computed. For a detailed description of the IvP solver solution algorithms, see [32].
36.1.6 Properties of the IvPDomain Class

The domain of an IvP function is an instance of the class IvPDomain. It is the same between all functions produced by all behaviors. The domain has a finite set of labeled variables with a lower and upper bound for each variable, and an integer number of evenly spaced points between the bounds. The domain is also referred to as the decision space. The 2D base of the cube in Figure 90 represents the domain. A point in the domain is contained in exactly one piece of an IvP function. The domain is built by the helm at the time of launch, and a copy is handed to each behavior in its constructor to ensure uniformity between behaviors. Listing 1 shows an example domain with three variables as it would be specified within the MOOS configuration block for the pHelmIvP process. See Listing 2.

Listing 36.1: An IvP domain with three variables, as specified in pHelmIvP configuration.

```plaintext
1 Domain = course:0:359:360
2 Domain = speed:0:3:16
3 Domain = depth:0:500:101
```

Each line augments the initially null domain with a new variable. The first of four arguments is the variable name, e.g., course. The second and third arguments indicate the lower and upper bound of the variable. They are integers here, but could be floating point values. The last of four arguments is the number of points in the domain for that variable. This domain would have \((360 \times 16 \times 101)\) 581,760 distinct possible decisions.

The behavior author, using the IvPBuild Toolbox, only needs to create a black-box function routine able to evaluate any point in the IvP domain with respect to the objectives of that behavior. To use the toolbox, this routine needs to reside within an implementation of a class that subclasses the AOF class described in Section 38.2. Although behaviors share a common domain, they can be defined over a different subset of variables as long as the common variables match in extents. For example, a behavior in a helm configured with the domain above could be defined over the following sub-domain:

```
Domain = depth:0:500:101
```

It could not be defined over:

```
Domain = depth:0:100:101
```

To facilitate the proper creation of sub-domains, the following function is provided in the build toolbox (in BuildUtils.h in lib_ivpbuild):

```
IvPDomain subDomain(IvPDomain original_domain, string variable_names);
```

The first argument is the original domain. The string argument is a comma separated list of variable names to be included in the sub-domain. If a variable is named in the argument that doesn’t exist in the original domain, an empty domain is returned. This is detected by checking the size of the domain with a call to `domain.size()`, which returns the number of domain variables. A proper sub-domain of the domain shown in Listing 1, with only the depth variable, could be created with the following function call:
#include "BuildUtils.h"
...
domain = subDomain(original_domain, "depth");

It is common for a behavior to declare its sub-domain in its constructor, even if it is expected to be
the same as the total helm domain. See for example line 21 in Listing.

A call to subDomain() has no effect if it names all of the original variables. By declaring a
sub-domain in the behavior’s constructor, problems can be avoided later if the overall helm domain
is expanded. If the function call above erroneously creates a null domain for the behavior, the helm
detects this automatically in the isRunnable() function call described in Section 7.4. This will cause
an error message to be posted to the BHV_ERROR variable and result in the helm posting all-stop
values to its actuators.

36.2 Tools Available in the IvPBuild Toolbox

The IvPBuild Toolbox contains a few different sets of tools. (a) The ZAIC tool is used for creating
IvP functions with only one decision variable. (b) The basic Reflector tool is used for creating IvP
functions over N coupled variables. (c) The advanced Reflector tools are an extension of the basic
Reflector tools that allow for piecewise defined functions with non-uniform pieces. (d) The Coupler
tool allows a pair of decoupled IvP functions to be converted to a single coupled IvP function. (e)
The encoding/decoding tools allow IvP functions to be converted to a string representation and
vice versa.

36.2.1 The ZAIC Tools for Functions with One Variable

The ZAIC tools are used for functions defined over a single decision variable. An example is shown
in Figure 91 where a fictional behavior may want to keep the vehicle in the so-called "deep sound
channel" or "SOFOR (SOund Fixing and Ranging) channel". This is a horizontal layer in the ocean
around which the speed of sound is at a minimum and where sound, especially at low frequencies,
may travel for thousands of meters with little loss of signal (See [?]).

Figure 91: An objective function with a single decision variable: This function assigns a maximum utility
to depths in a range of roughly 100 ± 20 meters. A linear decrease in utility is associated with depths outside this
interval up to another additional 20 meters.
A piecewise defined IvP function can be constructed to represent this function using five pieces. Assuming the "depth" decision space is 0 to 200 meters at one meter increments, the intervals would be: [0,59], [60,79], [80,120], [121,140], [141,200]. The linear function for each piece also needs to be set. This is not terribly difficult, but it is tedious and prone to human error. Instead, the ZAIC_PEAK utility is a tool in the ZAIC toolbox used for automating the production of IvP piecewise functions of the form shown in Figure 91. It is described in Section 37.

36.2.2 The Reflector Tool for Functions with Multiple Variables

The Reflector tool is used for creating IvP functions over n decision variables where n is greater than two. The Reflector was used to generate the IvP function rendered in Figure 90. The tools do work for n=1, but one variable functions are typically handled with the ZAIC tools described above. The Reflector produces an IvP function approximation of a given underlying function, where the underlying function is provided to the Reflector in the form of an instance of a class containing the underlying function implementation, and a specification of the IvP domain. The basic use of the Reflector is described in detail in Section 38, but the basic usage boils down to the following:

- Create an instance of the underlying function to be approximated.
- Create an instance of the Reflector, passing it the underlying function.
- Invoke the Reflector with a requested number of pieces.
- Retrieve the new IvP function from the Reflector.

The basic usage of the Reflector involves only the choosing the number of pieces used in the IvP function representation. By choosing only the number, pieces of uniform size will be used in the function. This suffices for most applications, but there are ways to produce a function that is both more accurate and uses less pieces by exploring advanced options and algorithms of the Reflector. These include:

- The Directed Refinement Reflector option.
- The Smart Refinement Reflector option.
- The AutoPeak Refinement Reflector option.

The details of these advanced options are discussed in Section 39. Each of the advanced tools are used after an initial basic uniform function has been generated. The directed refinement option allows the user to specify subsets of the domain and use pieces of different sizes for that region only. The smart refinement option asks the Reflector to estimate the fit of each piece as it is generated in terms of accuracy in approximating the underlying function and performs further refinement on those pieces that need it the most. The autopeak refinement option repeatedly refines the single piece containing the maxima of the underlying function until that point is contained in a piece containing only that point.

36.2.3 The Coupler Tool for Coupling Two Decoupled IvP Functions

Two IvP functions defined over different variables can be combined to form a single IvP function defined over the union of the two sets of variables. The basic usage of the Coupler can be summarized as follows:
• Create the two independent IvP functions.
• Create an instance of the OF_Coupler class.
• Pass the two functions to the Coupler.
• Retrieve the new IvP function from the Coupler.

This tool was used in the example SimpleWaypoint behavior of Section 40 in Listing 5 to couple two one-variable IvP functions. When a Coupler is passed the two IvP function pointers, it takes over ownership of the functions, i.e., it deletes the two one-variable functions when the Coupler object is deleted. When a coupled IvP function is extracted from the Coupler, ownership of the IvP function is passed to the caller, i.e., the caller is responsible for deleting the IvP function.
37 The ZAIC Tools for Building One-Variable IvP Functions

The ZAIC tools are part of the IvP Build Toolbox for facilitating the building of IvP functions over a single domain variable. There are three tools - ZAIC_PEAK, ZAIC_HEQ, ZAIC_LEQ, and ZAIC_Vector. To use the tools, in short, one creates an instance of the corresponding class, sets some parameters, and then extracts an IvP function. The tools described in Section 38 for n-variable functions can also be used for building one-variable functions but are perhaps overkill for certain classes of common one-variable functions that motivated the ZAIC tools. The term ZAIC is not an acronym, but merely a play on the word mosaic.

37.1 The ZAIC_PEAK Tool

37.1.1 Brief Overview

The ZAIC_PEAK tool is designed with the objective function shown in Figure 92 in mind. There is an identifiable preferred single decision choice (the summit) with maximum utility, and then a gradual drop in utility as the variable value varies from the preferred choice.

![Figure 92: The ZAIC_PEAK tool](image)

The form in which the utility drops is dependent on the settings of the six parameters shown in the figure. The summit, peakwidth, and basewidth values are given in units native to the decision variable, while the summitdelta, minutil, and maxutil values are given in terms of units of utility.

37.1.2 The ZAIC_PEAK Parameters and Function Form

The ZAIC_PEAK tool accepts six parameters in defining $f(x)$. The summit parameter is the point of maximum utility. The minutil parameter is the minimum value of $f(x)$, with a default value of zero.
The maxutil is the maximum value of $f(x)$, with a default value of 100. The utility of the function drops off linearly in two stages. In the first stage the utility drops linearly off from the maxutil to maxutil-summitdelta, and in the second stage it drops off linearly from maxutil-summitdelta to minutil. The function has the form:

$$f(x) = \begin{cases} 
  f_1(x) & (\text{summit} - \text{peakwidth}) \leq x \leq \text{summit}, \\
  f_2(x) & (\text{summit} - \text{peakwidth} - \text{basewidth}) \leq x < (\text{summit} - \text{peakwidth}), \\
  f_3(x) & \text{summit} < x \leq (\text{summit} + \text{peakwidth}), \\
  f_4(x) & (\text{summit} + \text{peakwidth}) < x \leq (\text{summit} + \text{peakwidth} + \text{basewidth}), \\
  \text{minutil} & \text{otherwise.} 
\end{cases}$$

where

$$f_1(x) = (\text{maxutil} - \text{summitdelta}) + (\text{summitdelta} \times ((x - (\text{summit} - \text{peakwidth})) / \text{peakwidth}))$$

$$f_2(x) = \text{minutil} + ((\text{maxutil} - \text{minutil} - \text{summitdelta}) \times ((x - (\text{summit} - \text{peakwidth} - \text{basewidth})) / \text{basewidth}))$$

$$f_3(x) = (\text{maxutil} - \text{summitdelta}) + (\text{summitdelta} \times ((\text{summit} + \text{peakwidth}) - x) / \text{peakwidth})$$

$$f_4(x) = \text{minutil} + ((\text{maxutil} - \text{minutil} - \text{summitdelta}) \times (((\text{summit} + \text{peakwidth} + \text{basewidth}) - x) / \text{basewidth}))$$

To correlate the above five cases above with the six pieces in Figure 92, $f_1(x)$ is piece #3, $f_2(x)$ is piece #2, $f_3(x)$ is piece #4, $f_4(x)$ is piece #5, minutil for pieces #1 and #6. The two stage linear drop-off in utility is there to allow the shape of the function to approximate convex or concave functions as shown in Figure 93.

![Figure 93: The ZAIC_Peak tool: A convex uni-modal function (left) and a non-convex uni-modal function (right).](image)

37.1.3 The ZAIC_Peak Interface Implementation

The following functions define the interface to the ZAIC_Peak tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were
detected. In such cases, a warning string is generated for optional retrieval, and the error renders
the instance effectively useless, never yielding an IvP function when requested.

Many of the below functions take an optional index parameter. This is used for creating functions
with multiple modes or peaks as described in Section 37.1.6. The default value is zero, or the
zeroth index, when only one mode or peak is being implemented. If the given index references a
non-existing component, this is considered a fatal configuration error. Example usage is provided in
Listing 1.

• bool setSummit(double val, int index=0): Sets the summit value of the component at the
given index. If no index parameter is provided, the index is zero. If the summit value is
outside the range of the domain, it is clipped to the appropriated end. For example if the
domain were $[0,359]$ as the example in Figure 92, and the requested summit value were 550, the
summit parameter would be set to 359. This is therefore not regarded as a fatal configuration
error, but a warning would be generated anyway. This returns false only if index referencing a
non-existent component is provided.

• bool setPeakWidth(double val, int index=0): Sets the peakwidth value of the component
at the given index. If no index parameter is provided, the index is zero.

• bool setBaseWidth(double val, int index=0): Sets the basewidth value of the component at
the given index. If no index parameter is provided, the index is zero.

• bool setSummitDelta(double val, int index=0): Sets the summitdelta value of the compo-
nent at the given index. If no index parameter is provided, the index is zero. A fatal error
is declared and false is returned if the index is out of range, or if the value is less than zero.
Otherwise true is returned. If the sumitdelta value is greater than the range determined by
maxutil - minutil, this is not interpreted as a fatal error, but the summitdelta is clipped to
the range.

• bool setMinMaxUtil(double min, double max, int index=0): Sets the minutil and maxutil
values of the component at the given index. If no index parameter is provided, the index is
zero. A fatal error is declared and false is returned if the index is out of range, or if the min
value is greater than or equal to the max value. Otherwise true is returned. If the existing
summitdelta value is greater than the range determined by maxutil - minutil, this is not
interpreted as a fatal error, but the summitdelta is clipped to the range.

• bool setParams(double summit, double peakwidth, double basewidth, double summitdelta,
double minutil, double maxutil, int index=0): Sets the six configuration parameters summit,
peakwidth, basewidth, summitdelta, minutil and maxutil all at once. If the summitdelta value
is greater than the range determined by maxutil - minutil, this is not interpreted as a fatal
error, but the summitdelta is clipped to the range.

• void setSummitInsist(bool val): Sets the summitinsist flag to the given value. The default
is true. See Section 37.1.4 for more on this parameter.

• void setValueWrap(bool val): Sets the valuewrap flag to the given value. The default is false.
See Section 37.1.4 for more on this parameter.

• int addComponent(): Allocates a new component and returns the index of the new component.
Since the first component exists upon ZAIC creation, the first call to this function will return 1,
and will result in the ZAIC instance having two components at index 0 and 1. The default
values for the new component are summit=0, peakwidth=0, basewidth=0, summitdelta=50,
minutil=0, maxutil=100.

- \textit{IvPFunction* extractIvPFunction(bool maxval=true)}: This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. If a fatal error was detected in prior parameter setting attempts, this function will simply return the NULL pointer. When the IvP function is extracted from the ZAIC, an \textit{IvPFunction} instance is created from the heap that needs to be later deleted. The ZAIC tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.

- \textit{string getWarnings()}: When or if fatal (or non-fatal) problems are encountered in setting the parameters, the tool appends a message to a local warning string. This string can be retrieved by this function. A non-empty string does not necessarily mean a fatal configuration error was encountered. Instead, the \textit{stateOK()} function below should be consulted.

- \textit{bool stateOK()}: This function returns true if no fatal errors were encountered during configuration attempts, otherwise it returns false. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the \textit{getWarnings()} function above can be consulted.

### 37.1.4 The Value-Wrap and Summit-Insist Parameters

Two additional Boolean parameters may be set for an instance of \texttt{ZAIC\_PEAK}. They are the \texttt{valuewrap} and \texttt{summitinsist} parameters set with the following functions with the defaults shown:

```c
zaic.setValueWrap(false); // Default value is false
zaic.setSummitInsist(true); // Default value is true
```

When the \texttt{valuewrap} parameter is true, the utility associated with the domain variable value "wraps" around. For example, if the domain variable is the vehicle \textit{heading}, which may have the domain \([0, 359]\), a value of 10 degrees is evaluated as being only 20 degrees different from 350, rather than being different by 330 degrees. The two functions depicted in Figure 94 differ only in the setting of the \texttt{valuewrap} parameter.

![Figure 94: The valuewrap parameter in the ZAIC\_PEAK tool: A function generated with valuewrap=false on the left, and function generated with valuewrap=true and otherwise identical parameters on the right.](image)
The value of the summitinsist parameter can affect the generated objective function in the following two scenarios. In the first case, consider the domain to be the possible headings with 360 discrete choices [0, 359], and the peakwidth and basewidth are both zero. If the summit were then set to 90.25, one way to interpret this is that all 360 discrete heading choices have a utility of zero, since none are equal exactly to 90.25. This is the interpretation when summitinsist is false. When set to true, the same set of parameters would generate an objective function that ranked the heading of 90 degrees with maximum utility and all other heading choices with the minimum utility (an IvP function with 3 pieces, i.e., intervals). In the second case, consider the domain to be possible speeds with 31 discrete choices [0, 3.0], with the summit set to 4.0 with a peakwidth and basewidth of 0.25. The ZAIC_PEAK tool would generate an objective function ranking all speeds equally with the minimum utility when summitinsist is set to false. When set to true, the ZAIC_PEAK tool would generate an objective function ranking the highest speed in the domain (3.0) with maximum utility (maxutil), and all other speeds with minimum utility (minutil). The default setting is summitinsist=true since this seems the more reasonable thing to do in most such cases.

37.1.5 Using the ZAIC_PEAK Tool

Usage of the ZAIC_PEAK tool boils down to the following four steps.

- Step 1: Create the IvP domain, or retrieve it if otherwise already created.
- Step 2: Create the ZAIC_PEAK instance with a domain and domain variable.
- Step 3: Set the ZAIC_PEAK parameters.
- Step 4: Extract the IvP function.

A code example of the four steps is provided in Listing 1 below. This code example describes a function that builds and returns an IvP function using the ZAIC_LEQ tool. It is not too different from the activity inside a typical implementation of onRunState in an IvP behavior.

Listing 37.1: Example usage of the ZAIC_Peak tool corresponding to Figure 92.

```c++
#include "ZAIC_PEAK.h"
...
1 IvPFunction *buildIvPFunction()
2 {
3   // Step 1 - Create the IvPDomain, the function’s domain
4   IvPDomain domain;
5   domain.addDomain("depth", 0, 600, 601);
6
7   // Step 2 - Create the ZAIC_PEAK with the domain and variable name
8   ZAIC_PEAK zaic_peak(domain, "depth");
9
10  // Step 3 - Configure the ZAIC_PEAK parameters
11  zaic_peak.setSummit(150);
12  zaic_peak.setMinMaxUtil(20, 120);
13  zaic_peak.setBaseWidth(60);
14
15  // Step 4 - Extract the IvP function
16  IvPFunction *ivp_function = 0;
17  if(zaic_peak.stateOK())
18     ivp_function = zaic_peak.extractIvPFunction();
19  else
20     cout << zaic_peak.getWarnings();
21  return(ivp_function)
```

284
The lines comprising step 4 (lines 15-20) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

    return(zaic_peak.extractIvPFunction());

This is simpler, but the warning information is potentially useful. When the ZAIC_PEAK tool is used within an IvP behavior, the warnings can be posted to the MOOSDB in the variable BHV_WARNING which is monitored by other tools.

### 37.1.6 Support for Multi-Modal Functions with the ZAIC_PEAK Tool

The ZAIC_PEAK tool will allow additional components to be added to provide a multi-modal effect. A *component* refers to the set of six parameters *summit*, *peakwidth*, *basewidth*, *summitdelta*, *minutil*, and *maxutil*. Adding a second component creates a multi-modal function as shown in Figure 95.

![Figure 95: Multiple modes with the ZAIC_PEAK tool: additional components can be added to create a multi-modal objective function. Each component is comprised of the six parameters summit, peakwidth, basewidth, summitdelta, minutil, and maxutil.](image)

Listing 2 shows how this is done. In lines 2-3, the ZAIC_PEAK instance is created and the parameters are set for the first component. A second component is allocated in line 5 with the call to `addComponent()` which returns the index of the newly created component. This index is passed as the last argument to the function call on line 6 to clarify that the parameters are to be applied to the second component (at index 1).

*Listing 37.2: Using the ZAIC_PEAK tool to build and return a multi-mode IvP Function.*

```cpp
#include "ZAIC_PEAK.h"
...

IvPFunction *buildIvPFunction(IvPDomain domain, string varname) {
    ZAIC_PEAK zaic(domain, varname);
    // Other code...
}
```

285
zaic.setParams(300, 80, 100, 15, 0, 100); // No index given - assumed to be zero.

int index = zaic.addComponent();

zaic.setParams(600, 130, 35, 30, 0, 147, index); // Last param is component index

zaic.setValueWrap(false); // Not component specific - no index given

zaic.setSummitInsist(true); // Not component specific - no index given

bool take_the_max = true

IvPFunction *ipf = zaic.extractIvPFunction(take_the_max);

return(ipf);
}

When the IvP function is extracted from the ZAIC_PEAK, in line 12, the composition of the multiple components can be interpreted in one of two ways - by taking the maximum or the sum of the two components. In the former, the combined utility is the maximum of the values given by the individual components. In the latter the combined utility is the sum of the individual components. The extractIvPFunction(bool) function on line 12 will return a composition based on taking the max if passed true, and will take the sum otherwise. The default (if no value is passed) is true. The difference between the two extractions is shown in Figure 96 for the two components shown in Figure 95.

Figure 96: Options for combining components: On the left is the result of combining the two components in Figure 95 by using the max value from the two components. On the right is the result of combining the two components by using the sum of the values from the two components.

All component parameter-setting functions defined for the ZAIC_PEAK class take an optional final argument indicating the intended component index. If the argument is not provided, it is assumed to be zero, the index of the one component created automatically when the ZAIC_PEAK instance is created. The default values of a newly added component are are summit=0, peakwidth=0, basewidth=0, summitdelta=50, minutil=0, maxutil=100. The valuewrap and summitinsist parameters are not component-specific parameters, and therefore are only called once on a particular ZAIC_PEAK instance, and do not have an optional index parameter.
37.2 The \texttt{ZAIC\_LEQ} and \texttt{ZAIC\_HEQ} Tools

37.2.1 Brief Overview

The \texttt{ZAIC\_LEQ} tool is used for generating IvP functions where there is a constant, maximum utility associated with a decision variable whose value is kept \textit{less than or equal} to (LEQ) a given value. For example if a UUV component is known to work reliably up to a certain depth, or if a vehicle's speed is to be kept below a certain value to prevent interference with another sensor or communications equipment. The \texttt{ZAIC\_LEQ} tool allows for expressing a linear drop-off in utility between two values. For example, if the fictional UUV component is rated to a depth of 200 meters, the utility function may have a maximum utility for depths up to 150 meters and minimum utility at 210 meters as in the example in Figure 97.

![Figure 97: The ZAIC\_LEQ tool](image)

Likewise, the \texttt{ZAIC\_HEQ} tool is used for generating objective functions where there is a constant, maximum utility associated with a decision variable whose value is kept \textit{greater than or equal} to (HEQ) a given value. These two tools have a similar interface. Most of what is described below for one tool applies to the other. The differences are distinguished in Section 37.2.5. The IvP functions generated by these ZAIC tools have a small footprint, having either two or three pieces.

37.2.2 The \texttt{ZAIC\_LEQ} Parameters and Function Form

The \texttt{ZAIC\_LEQ} tool accepts four parameters in defining $f(x)$. The \texttt{summit} parameter is the point where maximum utility begins to drop off. The \texttt{minutil} parameter is the minimum value of $f(x)$, with a default value of zero. The \texttt{maxutil} is the maximum value of $f(x)$, with a default value of 100. The function has the form:
The `basewidth` parameter can be used to soften the drop in utility as shown in Figure 97. When `basewidth` has the default value of zero, the general form is as above. When `basewidth` is configured with a positive value, the general form is:

\[
  f(x) = \begin{cases} 
  \text{maxutil} & x \leq \text{summit}, \\
  \text{minutil} + ((\text{maxutil} - \text{minutil}) \times ((x - \text{summit}) / \text{basewidth})) & \text{summit} < x \leq \text{summit} + \text{basewidth}, \\
  \text{minutil} & \text{otherwise}. 
  \end{cases}
\]

For the example in Figure 97, the function is given below.

\[
  f(x) = \begin{cases} 
  120 & x \leq 150, \\
  20 + ((120 - 20) \times ((210 - x)/(210 - 150))) & 150 < x \leq 210, \\
  20 & \text{otherwise}. 
  \end{cases}
\]

The three above cases corresponds to the three pieces generated for the IvP function shown in Figure 97.

### 37.2.3 The ZAIC_LEQ Interface Implementation

The following functions define the interface to the ZAIC_LEQ tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested. Example usage is provided in Listing 3.

- **ZAIC_LEQ(IvPDomain domain, string varname):** The constructor takes two arguments, an IvP domain and a variable name contained in the domain. The named variable needs to be just one of the variables used in the IvP domain; not necessarily the only one. If the named variable is not part of the IvP domain, this is regarded as a fatal error.
- **bool setSummit(double summit):** Sets the `summit` value. If the summit value is outside the range of the domain, it is clipped to the appropriated end. For example if the domain were [0,600] as the example in Figure 97, and the requested summit value were 650, the summit parameter would be set to 600. This is therefore not regarded as a fatal configuration error, but a warning would be generated anyway. This function always returns true.
• **bool setMinMaxUtil(double min, double max):** Sets the values for minutil and maxutil. If the minutil is greater or equal to maxutil, this is regarded as a fatal configuration error, a warning is generated, and the function returns false.

• **bool setBaseWidth(double basewidth):** Sets the basewidth parameter value. The given value must be greater than or equal to zero. Otherwise this is regarded as a fatal configuration error, a warning is generated, and the function returns false.

• **IvPFunction *extractIvPFunction():** This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. If a fatal error was detected in prior parameter setting attempts, this function will simply return the NULL pointer. When the IvP function is extracted from the ZAIC, an IvPFunction instance is created from the heap that needs to be later deleted. The ZAIC tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.

• **string getWarnings():** When or if fatal (or non-fatal) problems are encountered in setting the parameters, the tool appends a message to a local warning string. This string can be retrieved by this function. A non-empty string does not necessarily mean a fatal configuration error was encountered. Instead, the stateOK() function below should be consulted.

• **bool stateOK():** This function returns true if no fatal errors were encountered during configuration attempts, otherwise it returns false. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the getWarnings() function above can be consulted.

37.2.4 Using the ZAIC_LEQ Tool

Usage of the ZAIC_LEQ tool boils down to the following four steps.

• Step 1: Create the IvP domain, or retrieve it if otherwise already created.
• Step 2: Create the ZAIC_LEQ instance with a domain and domain variable.
• Step 3: Set the ZAIC_LEQ parameters.
• Step 4: Extract the IvP function.

A code example of the four steps is provided in Listing 3 below. This code example describes a function that builds and returns an IvP function using the ZAIC_LEQ tool. It is not too different from the activity inside a typical implementation of onRunState in an IvP behavior.

Listing 37.3: Example usage of the ZAIC_LEQ tool corresponding to Figure 97.

```cpp
#include "ZAIC_LEQ.h"
...
IvPFunction *buildIvPFunction()
{
    // Step 1 - Create the IvPDomin, the function's domain
    IvPDomain domain;
    domain.addVar("depth", 0, 600, 601);

    // Step 2 - Create the ZAIC_LEQ with the domain and variable name
    ZAIC_LEQ zaic_leq(domain, "depth");
}```
// Step 3 - Configure the ZAIC_LEQ parameters
zaic_leq.setSummit(150);
zaic_leq.setMaxMinUtil(20, 120);
zaic_leq.setBaseWidth(60);

// Step 4 - Extract the IvP function
IvPFunction *ivp_function = 0;
if(zaic_leq.stateOK())
  ivp_function = zaic_leq.extractIvPFunction();
else
  cout << zaic_leq.getWarnings();
return(ivp_function)

The lines comprising step 4 (lines 15-20) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

    return(zaic_leq.extractIvPFunction());

This is simpler, but the warning information is potentially useful. When the ZAIC_LEQ tool is used within an IvP behavior, the warnings can be posted to the MOOSDB in the variable BHV_WARNING which is monitored by other tools.

37.2.5 The ZAIC_HEQ Tool

Like the ZAIC_LEQ tool, the ZAIC_HEQ too is used for generating objective functions where there is a constant, maximum utility associated with a decision variable whose value is kept greater than or equal to a given value. The parameters described in Section 37.2.2 and interface implementation described in Section 37.2.3 for the ZAIC_LEQ tool are identical for the ZAIC_HEQ tool. The parameters are interpreted differently however. The same parameters used in Figure 97 are used in the ZAIC_HEQ tool to give the function shown in Figure 98.
Figure 98: The ZAIC\_LEQ tool: facilitates building simple 2-3 piece piecewise defined utility functions over a single decision variable whose value is to be kept greater than or equal to a given value. It accepts four parameters, the summit, minutil, maxutil, basewidth. In the figure summit=150, minutil=20, maxutil=120, basewidth=60.

37.2.6 A Warning about the Maximum Utility Plateau

It is worth noting a potential pitfall regarding the maximum utility plateau generated by both the ZAIC\_LEQ and ZAIC\_HEQ tools. By having equal utility for all domain values in the plateau range, no one domain value is preferred. If this is the only IvP objective function involved in the decision process for the particular domain variable, it is not clear what value will be chosen by the IvP solver. Even more troublesome is that the chosen value may change between iterations giving the appearance that the decision engine is thrashing. In this case the solver is simply faithfully and strictly interpreting the problem it was given. In short, these objective functions are designed to work in conjunction with others that express preferences in a non-plateau manner.

37.3 The ZAIC\_Vector Tool

37.3.1 Brief Overview

The ZAIC\_Vector tool is used for generating IvP functions over one variable, given some number of explicit domain-range mappings. These mappings are passed to the ZAIC\_Vector tool as two equally sized vectors; a vector of domain values and a vector of range values. An IvP function is created that typically has a piece per given domain-range pair, where the slope of each piece simply approximates the domain-range characteristics for domain-range pairs not explicitly given.
37.3.2 Using the ZAIC_Vector Tool

Usage of the ZAIC_Vector tool boils down to the following four steps.

- **Step 1**: Create the IvP domain, or retrieve it if otherwise already created.
- **Step 2**: Create the ZAIC_Vector instance with a domain and domain variable.
- **Step 3**: Set the ZAIC_Vector parameters.
- **Step 4**: Extract the IvP function.

A code example of the four steps is provided in Listing 4 below. This code example describes a function that builds and returns an IvP function using the ZAIC_Vector tool. It is not too different from the activity inside a typical implementation of `onRunState` in an IvP behavior.

**Listing 37.4: Example usage of the ZAIC_Vector tool corresponding to Figure 99.**

```cpp
#include "ZAIC_Vector.h"
...
IvPFunction *buildIvPFunction()
{
    // Step 1 - Create the IvPDomain, the function's domain
    IvPDomain domain;
    domain.addVar("depth", 0, 600, 601);

    // Step 2 - Create the ZAIC_Vector with the domain and variable name
    ZAIC_Vector zaic_vector(domain, "depth");

    // Step 3 - Configure the ZAIC_Vector parameters
    vector<double> domain_vals;
    vector<double> range_vals;
    domain_vals.push_back(100); range_vals.push_back(80);
    //...
```
The lines comprising step 4 (lines 26-31) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

```c
return(zaic_vector.extractIvPFunction());
```

This is simpler, but the warning information is potentially useful. When the ZAIC_LEQ tool is used within an IvP behavior, the warnings can be posted to the MOOSDB in the variable `BHV_WARNING` which is monitored by other tools.
38 The Reflector Tool for Building N-Variable IvP Functions

38.1 Overview

The IvPBuild Toolbox contains the Reflector tool for building IvP functions over \( n \geq 2 \) decision variables. Although the tools work with \( n = 1 \) variables, the ZAIC tools are typically used instead. The Reflector tool operates on a particular division of labor. The user of the Reflector provides a black-box function implementation able to provide a utility value for any queried input. The possible queries are limited to the domain or decision space of the function expressed with an IvPDomain instance. This black-box routine in essence is the underlying objective function to be approximated by the generated IvP function (Figure 100).

![Figure 100: The Reflector Tool: An IvP function approximates an underlying function \( f(x,y) \) using a piecewise linear structure with 698 pieces. The piece distribution need not be uniform allowing greater resolution over parts of the domain where the function is detected to be less locally linear.](image)

The goal is to generate an acceptable IvP function approximation by querying the underlying function for as a small subset of the total function domain as possible. The CPU time taken to evaluate the underlying function can easily be the most expensive part of building the IvP function for a given behavior. Implementing the underlying function efficiently and in a way that accurately reflects the intent of the behavior can be the most challenging part of building a behavior. The Reflector tool can be used with a very simple interface that builds an IvP function given a pointer to the underlying function and the number of pieces to use in the IvP function. Such IvP functions will be constructed with pieces of uniform size. This is discussed in Section 38.3. The Reflector can be configured with more advanced parameters to build an IvP function with non-uniformly distributed pieces as depicted in Figure 100. These methods are used to build functions that more accurately approximate their underlying functions and use less pieces. The advanced Reflector parameters are discussed in Section 39.
38.2 Implementing Underlying Functions within the AOF Class

The primary job of a behavior author is to provide a method capable of evaluating any candidate
decision in the decision space. Evaluating each decision can be prohibitively time consuming and a
piecewise linear approximation with an IvP function is typically built by invoking the evaluation
function for only a small subset of the domain. The build toolbox depends on access to the evaluation
routine in a generic way, as a pointer to an instance of the class AOF, the "actual objective function".

38.2.1 The AOF Class Definition

The AOF class itself is abstract, and the AOF pointer actually points to an implemented subclass
with a few key virtual functions overloaded. The AOF class definition (slightly simplified) is given in
Listing 1.

Listing 38.1: The AOF class definition.

```cpp
#include "IvPBox.h"
#include "IvPDomain.h"

class AOF{
public:
  AOF(IvPDomain domain) {m_domain=domain;};
  virtual ~AOF() {};
  virtual double evalPoint(vector<double>); 
  virtual bool setParam(string, double) {return(false);};
  virtual bool setParam(string, string) {return(false);};
  virtual bool initialize() {return(true);};
  double extract(string, const vector<double>&) const;
protected:
  IvPDomain m_domain;
};
```

This is essentially a template for a function, defined over the domain given in the constructor. The
mapping from the domain to a range is implemented in the evalPoint() function which takes a
vector of numerical values representing a candidate decision in the IvP domain decision space. The
setParameter() and initialize() virtual functions provide a generic way for subclasses to set their
parameters.

38.2.2 An Example Underlying Function Implemented as an AOF Subclass

As an example consider the simple linear function \( f(x,y) = mx + ny + b \), implemented by the
class AOF_Linear shown in Listing 2 and 3 below. The class contains three member variables, lines
11-13 in Listing 2 for representing the coefficient and scalar parameters.

Listing 38.2: AOF_Linear.h - The class definition for the AOF_Linear class.

```cpp
#include "AOF.h"
class AOF_Linear: public AOF {
public:
  AOF_Linear(IvPDomain domain) : AOF(domain) 
  {m_coeff = 0; n_coeff=0; b_scalar=0;};
  "AOF_Linear() {};
```
In lines 4-5 above, the constructor takes an instance of IvPDomain as an argument and passes it to the AOF superclass for handling (line 4). The remainder of the class implementation is shown in Listing 3. The m and n coefficients are set in the setParam() function and will return true if either the m_coeff, n_coeff, or b_scalar parameters are passed, and false otherwise.

Listing 38.3: AOF_Linear.cpp - The class implementation for the AOF_Linear class.

```cpp
1 double evalPoint(vector<double>);
2 bool setParam(const string& param, double val);
3
4 private:
5 double m_coeff;
6 double n_coeff;
7 double b_scalar;
8 );

11 //-------------------------------------------------------------------------------
12 bool AOF_Linear::setParam(string param, double val)
13 {
14 if(param == "mcoeff")
15     m_coeff = val;
16 else if(param == "ncoeff")
17     n_coeff = val;
18 else if(param == "bscalar")
19     b_scalar = val;
20 else
21     return(false);
22 return(true);
23 //-------------------------------------------------------------------------------
24 double AOF_Linear::evalPoint(vector<double> point)
25 {
26 double x_val = extract("x", point);
27 double y_val = extract("y", point);
28 return((m_coeff * x_val) + (n_coeff * y_val) + b_scalar);
29 }
```

The evalPoint() function in lines 16-22 is where the actual implementation of \( f(x,y) = m \cdot x + n \cdot y + b \) is implemented (on line 21). The argument to this function is a vector of values holding the values for the \( x \) and \( y \) variables. The ordering of these values i.e., which of the two variables, \( x \) or \( y \), is contained in the first value of the vector, is sorted out in the two calls to the extract() function in lines 18-19. This sorting out is possible because the ordering is determined by the IvPDomain member variable defined at the AOF superclass level, and provided in the constructor. The AOF_Linear class, used as an example here, is included in the code distribution in lib_ivpbuild. It may serve as a template in building a new AOF_YourAOF class. It also can be used to verify that the build tools will work in the extreme case of creating a piecewise function with only one piece. And in the case of AOF_Linear the piecewise ”approximation” is exact.

38.2.3 Another AOF Example Class Implementation for Gaussian Functions

A second function type, implementing Gaussian functions, is implemented as the AOF_Gaussian class in the IvP Toolbox in the lib_ivpbuild module. This function is a bit more interesting in that
a piecewise linear approximation needs multiple pieces to generate a fairly good approximation (understanding that "fairly good" is subjective). It is also interesting in that, depending on the configuration, there may be large portions of the function that are indeed locally linear and in need of relatively few pieces to generate a decent approximation. This function will be used extensively in later examples of usage and performance of the Reflector tool. The Gaussian function form is given by:

\[ f(x, y) = Ae^{-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}} \]

The function is defined over the two variables, \( x \) and \( y \), and has four parameters. Examples for two groups of parameter settings are shown in Figure 101, and in Figure 102. The coefficient, \( A \), is the amplitude, \( x_0 \) and \( y_0 \) are the center, and \( \sigma \) represents the spread of the blob. This function is implemented by the class \texttt{AOF\_Gaussian} shown in Listings 4 and 5. Note that it is a subclass of the \texttt{AOF} class and overrides the critical function \texttt{evalPoint()}. It also implements the \texttt{setParam()} function for setting the four parameters in the above equation.

\textbf{Listing 38.4: The class definition for the AOF\_Gaussian class.}

```cpp
1 class AOF_Gaussian: public AOF {
2     public:
3         AOF_Gaussian(IvPDomain domain) : AOF(domain) {
4             m_xcent=0; m_ycent=0; m_sigma=1; m_range=100;};
5     ~AOF_Gaussian() {}
6     double evalPoint(vector<double> point);
7     bool setParam(string param, double value);
9
10     private:
11         double m_xcent;
12         double m_ycent;
13         double m_sigma;
14         double m_range;
16     };
```

\textbf{Listing 38.5: The class implementation for the AOF\_Gaussian class.}

```cpp
1 //************************************************************************
2 // Procedure: setParam
3
4 bool AOF_Gaussian::setParam(string param, double value)
5 {
6     if(param == "xcent") m_xcent = value;
7     else if(param == "ycent") m_ycent = value;
8     else if(param == "sigma") m_sigma = value;
9     else if(param == "range") m_range = value;
10     return(true);
11 }
13
15 //************************************************************************
16 // Procedure: evalPoint
17
18 double AOF_Gaussian::evalPoint(vector<double> point)
```

297
An example is shown in Figure 101 below. The domain for both the x and y variables is \([-250, 250]\) containing 501 x 501 = 251,001 points.

Figure 101: A Gaussian function: A rendering of the function $f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where $A = \text{range} = 100$, $\sigma = \text{sigma} = 150$, $x_0 = xcent = 0$, $y_0 = ycent = 0$. The domain for $x$ and $y$ ranges from $-250$ to $250$.

### 38.3 Basic Reflector Tool Usage Tool with Examples

Using the Reflector tool boils down to the four steps below. The third step may be non-existent if the user is building simple uniform functions.

- **Step 1:** Create the underlying function, AOF instance, and set its parameters.
- **Step 2:** Create the `Reflector` instance passing it a pointer to the AOF instance.
- **Step 3:** Set parameters for the `Reflector` if necessary or desired.
- **Step 4:** Direct the `Reflector` to build the IvP function and then extract it.

A code example of the four steps is provided in Listing 6 below. This code example describes a function that builds and returns an IvP function using the `Reflector` tool. It is not too different from the activity inside a typical implementation of `onRunState()` in an IvP behavior.
Listing 38.6: An example use of the Reflector to create a uniform IvP function.

```cpp
IvPFunction *buildIvPFunction(IvPDomain ivp_domain) {
  // Step 1 - Create the AOF instance and set parameters
  AOF_Gaussian aof(ivp_domain);
  aof.setParam("xcent", 50);
  aof.setParam("ycent", -150);
  aof.setParam("sigma", 32.4);
  aof.setParam("range", 150);

  // Step 2 - Create the Reflector instance given the AOF
  OF_Reflector reflector(&aof);

  // Step 3 - Parameterize the Reflector (None in this case)

  // Step 4 - Build and Extract the IvP Function
  int amt_created = reflector.create(1000);
  IvPFunction *ipf = reflector.extractIvPFunction();

  cout << "Pieces in the new IvPFunction: " << amt_created << endl;
  return(ipf);
}
```

The underlying function is created on lines 3-7 creating the Gaussian function with parameters shown in Figure 102. The Reflector is created on line 10 with a pointer to the new Gaussian underlying function. In lines 15-16, the Reflector creates and returns the IvP function.

![Gaussian function image](image)

Figure 102: A Gaussian function: A rendering of the function \( f(x,y) = Ae^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}} \) where \( A = \text{range} = 150 \), \( \sigma = \text{sigma} = 32.4 \), \( x_0 = \text{xcent} = 50 \), \( y_0 = \text{ycent} = -150 \). The domain for \( x \) and \( y \) ranges from \(-250\) to \(250\).

In this simple style of usage, no parameters are set on the Reflector after it is created. The result will be an IvP function with uniform piece shape, where the total number of pieces are
requested on line 15. (Note that 1000 pieces are requested, but not all requested piece counts are feasible or practical. See Section 39.2.2 for more on this). The requested number of uniform pieces affects three practical metrics of the resulting the IvP function. The error in its representation of the underlying function, the time to create the IvP function, and the number of pieces in the IvP function. The goal is to minimize each, but they are in competition with each other.

Figure 103 depicts four IvP function approximations of the same underlying function, and Table 60 illustrates the relationship between the three metrics of (a) piece count, (b) create time, and (c) accuracy in representing the underlying function. The user determines the most appropriate compromise between these metrics for the application at hand. In general, a gain on one metric is traded off against a sacrifice on other metrics. With the additional tools described in Section 39, it is often possible to make improvements in all three metrics simultaneously. One way to look at this is that there is a fourth metric, ease-of-use, that can instead be dialed back to achieve gains in all of the first three metrics. In Listing 6, the absence of Step 3, where insightful parameters could have been provided to the Reflector to produce non-uniform functions, could be viewed as optimizing the ease-of-use metric.

Figure 103: Four IvP functions approximating the same underlying function: Each IvP function uses a different number of uniform pieces.
<table>
<thead>
<tr>
<th>Case</th>
<th>Edge Size</th>
<th>Pieces</th>
<th>Layout</th>
<th>Worst Error</th>
<th>Avg Error</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>27889</td>
<td>(167x167)</td>
<td>0.0761</td>
<td>0.0014</td>
<td>656.4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1000</td>
<td>(100x100)</td>
<td>0.3019</td>
<td>0.0048</td>
<td>160.0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>5184</td>
<td>(72x72)</td>
<td>0.6720</td>
<td>0.0104</td>
<td>83.3</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2500</td>
<td>(50x50)</td>
<td>1.4589</td>
<td>0.0232</td>
<td>39.9</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>1156</td>
<td>(34x34)</td>
<td>3.4532</td>
<td>0.0551</td>
<td>18.9</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>625</td>
<td>(25x25)</td>
<td>5.8555</td>
<td>0.1014</td>
<td>10.4</td>
</tr>
<tr>
<td>7</td>
<td>25</td>
<td>400</td>
<td>(20x20)</td>
<td>7.79764</td>
<td>0.1585</td>
<td>6.5</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>289</td>
<td>(17x17)</td>
<td>12.0347</td>
<td>0.2303</td>
<td>4.7</td>
</tr>
<tr>
<td>9</td>
<td>40</td>
<td>169</td>
<td>(13x13)</td>
<td>24.2977</td>
<td>0.3919</td>
<td>2.8</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>100</td>
<td>(10x10)</td>
<td>18.2113</td>
<td>0.5917</td>
<td>1.6</td>
</tr>
<tr>
<td>11</td>
<td>75</td>
<td>49</td>
<td>(7x7)</td>
<td>42.0652</td>
<td>1.2143</td>
<td>0.9</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>25</td>
<td>(5x5)</td>
<td>30.3938</td>
<td>2.0285</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 60: IvP function configurations and metrics: The relationship between piece size, accuracy and construction time is shown for varying uniform piece size. Four of the row entries are rendered in Figure 103.

### 38.4 The Full Reflector Interface Implementation

The following functions define the interface to the Reflector tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested. Example usage is provided in Listing 6.

- **OF_Reflector(AOF*)**: The constructor takes a single argument, a pointer to the underlying function to be approximated by the Reflector. The AOF instance contains an instance of the IvPDomain which will also be the IvPDomain of any IvP functions created with the Reflector.
- **int create(int pieces=-1)**: This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. Many of the parameters affecting the form of the function are settable separately in the `setParam()` function, including the parameter specifying the number of pieces. If the optional `pieces` argument is provided in this function call, and if the value of the argument is ≥ 1, this overrides any piece count request set otherwise. This function will create an IvP function that the user can then obtain via the function `extractIvPFunction()` described below. The integer value returned is the number of pieces in the newly created IvP function. A value of zero indicates something has gone wrong.
- **IvPFunction *extractIvPFunction()**: This function returns a new IvP function built during a prior invocation of the `create()` function described above. If an error was encountered in either the parameter setting attempts, or in the invocation of the `create()` function, this function will simply return the NULL pointer. When the IvP function is extracted from the Reflector, an IvPFunction instance is created from the heap that needs to be later deleted. The Reflector tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.
- **string getWarnings()**: When or if problems are encountered in setting the parameters, the Reflector appends a message to a local warning string. This string can be retrieved by this
function.

• \texttt{bool stateOK()}: This function returns true if no errors were encountered during configuration attempts, otherwise it returns false. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the \texttt{getWarnings()} function above can be consulted.

• \texttt{bool setParam(string param, string value)}: This function is used for setting parameters on many optional tools more advanced than specifying the number of pieces to be used in a simple uniform function. An overview is provided here, with more detailed deferred to later sections that cover the advanced tools.

• \texttt{uniform_amount}: The amount of pieces to use in the creation of a simple uniform function. Alternatively can be supplied in the call to \texttt{create()} as described above.

• \texttt{uniform_piece}: A string description of the size and shape of a piece used during the creation of a pure uniform function. Details described in Section 39.1.2.

• \texttt{strict_range}: When set to true, the range of the linear interior function is guaranteed to stay within the range of any sampled points of the underlying function, even if a better overall fit could be obtained otherwise. The default is true.

• \texttt{refine_region}: A string description of a region of the IvP domain within which further directed refinement is requested. See Section 39.1.3.

• \texttt{refine_piece}: A string description of the size and shape of uniform pieces to be used within a region of directed refinement. See Section 39.3.

• \texttt{refine_point}: A string description of a point within the IvP domain to direct further refinement. See Section 39.3.

• \texttt{smart_amount}: The number of pieces to use in the smart refinement algorithm, beyond the number of pieces used in an initial simple uniform function. See Section 39.4.

• \texttt{smart_percent}: The number of pieces to use in the smart refinement algorithm specified as a percentage of the number of pieces used in an initial simple uniform function. See Section 39.4

• \texttt{smart_thresh}: A threshold given in terms of worst noted error between the IvP function and the underlying function, below which the smart refinement algorithm will cease further refinement. See Section 39.4.

• \texttt{auto_peak}: Set to either true or false indicating whether the auto-peak algorithm should be applied. See Section 39.5.

• \texttt{auto_peak_max_pcs}: The maximum amount of new pieces added to an IvP function during the auto-peak heuristic. See Section 39.5.

• \texttt{bool setParam(string param, double value)}: The parameters that may be set via this function may also be set via the \texttt{setParam()} function above where the \texttt{value} parameter is a string. This alternate method is implemented solely as a convenience to the caller.

• \texttt{uniform_amount}: See above.

• \texttt{smart_amount}: See above.

• \texttt{smart_percent}: See above.

• \texttt{smart_thresh}: See above.

• \texttt{auto_peak_max_pcs}: See above.
39 Optional Advanced Features of the Reflector Tool

39.1 Preliminaries

The previous section discussed how to build IvP functions with the Reflector tool in the IvP Build Toolbox by simply specifying a desired number of pieces in the resulting piecewise defined function. This section discusses a few further methods for building functions that give the user more control of the build process and typically better overall results in terms of fewer pieces, less time to build, and greater accuracy in the piecewise approximation of the underlying function.

39.1.1 The Reflector-Script

The basic invocation of the Reflector create() function may take a single argument requesting the number of pieces to be used in the piecewise function. An example is line 15 in Listing 6. In reality the invocation of create() is comprised of a script of distinct build heuristics of which the creation of uniform sized pieces is just the first of four parts. The latter three parts are optional and require further user configuration before being included for execution in the script. The four parts are:

- Uniform function creation
- Directed refinement
- Smart refinement
- Auto-Peak refinement

These four heuristics are discussed in the next four sections. Uniform function creation is revisited since finer control can be used (with typically better results) if the choice of piece size and shape is not left to the heuristic that converts the requested total number of pieces into an actual uniform piece shape.

39.1.2 Specifying a Piece Shape or IvP Domain Point in String Format

Aspects of the Reflector tool require the specification of the shape of a piece used in a piecewise defined IvP function. The specification is comprised of the length of the piece for each of the $n$ dimensions, i.e., decision variables. There are two ways to describe the lengths. Recall that the IvP domain for a variable is given by a low and high value, and the number of points. For example the variable $x$ could range from 0 to 30 with 31 points, and $y$ could range from $-50$ to 50 with 21 points. The first way to describe the length of a piece is by specifying the number of discrete points:

"discrete @ x:5,y:5"

A uniform function built over this domain with the above requested piece shape would have 35 pieces in a manner rendered in Figure 104.
Figure 104: A uniform IvP function: An IvP domain is rendered over the two variables $x$, with 31 elements, and $y$ with 21 elements. Requesting a set of uniform pieces with five elements on each edge results in the piece distribution shown. The circled point represents the 23rd index into the $x$ domain and the 13th index into the $y$ domain. This point can be referenced by the string "discrete @ x:22,y:12". It may also be referenced by the string "native @ x:22,y:10".

Note the distribution of pieces is not completely uniform. Smaller pieces are used at the upper ranges of the domain. A second method of specifying the same piece shape is to use the native lengths of the domain:

"native @ x:5,y:25"

This piece also has a length of five units along the $x$ dimension and five units along the $y$ dimension, resulting in the same distribution shown in Figure 104. When a “native” value doesn’t exactly map onto one of the points in the domain, it is rounded to the nearest domain point. For example, "native @ x:5,y:22.6" specifies a piece with five units on the $y$ dimension, "native @ x:5,y:22.4" specifies a piece with four units on the $y$ dimension. And when a native value is given exactly between two domain points, the value is rounded up, so "native @ x:5,y:22.5" specifies a piece with five units on the $y$ dimension.

A single point in the IvP domain can be similarly referenced. When the string "discrete @ x:5,y:5" is used to represent a piece shape, the numerical values represent the length of the piece. When the same string is used to represent a point in the IvP domain, the numerical values represent the index into the domain. For example, the circled point in Figure 104 can be be referenced by the string "discrete @ x:22,y:12". It may also be referenced by the string "native @ x:22,y:10". When a native values does not map exactly to a domain value, the nearest domain point is used.
39.1.3 Specifying a Region of an IvP Domain in String Format

Aspects of the \texttt{Reflector} tool require the specification of a region of the IvP domain. The specification is comprised of an upper and lower bound for each of the \( n \) dimensions, i.e., decision variables. Recall that the IvP domain for a variable is given by a low and high value, and the number of points. For example the variable \( x \) could range from 0 to 30 with 31 points, and \( y \) could range from \(-50\) to 50 with 21 points. A region can be specified as follows:

"native \texttt{@ x:10:24,y:-25:20}"

or equivalently,

"discrete \texttt{@ x:10:24,y:5:14}"

This region is rendered in Figure 105. If the extents specified in the string exceed the boundaries of the IvP domain, the requested region is clipped to be exactly the boundary value. For example, the string "\texttt{native @ x:10:24, y:-25:50}" and "\texttt{native @ x:10:24, y:-25:50000}" would specify the same region given the example in Figure 105.

When a native value is specified that does not map to a domain value, this case is handled differently for \texttt{regions} than it was when specifying a piece \texttt{shape}. In a region specification the native value is treated as a strict boundary value. Therefore the string "\texttt{native @ x:9.01:24.99, y:-29.99:24.99}" would specify the exact same region as the example above and in Figure 105.
39.2 Optional Feature #1: Choosing the Piece Shape in Uniform Functions

39.2.1 Potential Advantages

By simply specifying the desired number of pieces, the Reflector heuristically sets the piece size and aspect ratio of an initial uniform function. This has the advantage of being very simple and independent of the underlying function. (See line 15 in Listing 6.) However, like most heuristics, there may be cases where the result may not be best for a particular situation. If the user has some insight into the underlying function and the IvP domain, the user may not wish to leave this decision to the heuristic, but instead specify the piece shape explicitly. Below, the piece count-to-piece shape heuristic is described as well as how to override the heuristic with an explicit shape request.

39.2.2 Specifying the Piece Shape Implicitly from a Piece Count Request

When the Reflector creates a uniform IvP function based on a requested piece count, a heuristic is invoked to generate a single piece to be used in the uniform function based on both the piece count and the IvP domain. This piece is not unlike the 5 x 5 piece in Figure 104, except that a 5 x 5 piece is not explicitly requested, but rather the total pieces in that figure, 35, would be requested. Knowing a little about this heuristic can help determine when its worth the effort to instead explicitly define the shape of the uniform piece. The total requested pieces is an upper limit, and often not exactly achieved. For example, the same 35 pieces in Figure 104 would be created upon piece-count requests of 35, 36, 37, 38, and 39 pieces. The heuristic attempts to keep the aspect ratio of the uniform piece close to 1.0, but will deviate to allow a uniform piece that will result in a total number of pieces closer to the requested amount. The heuristic is given Listing 1 below, and some examples are shown in Table 61.

Listing 39.1: The heuristic for generating a uniform piece based on piece-count and domain.

```cpp
IvPBox buildUniformPiece(IvPDomain domain, int max_amount) {
    int dim = domain.getDim();
    vector<int> pcs_on_edge(dim,1);
    vector<bool> pcs_maxed(dim,false);
    vector<int> pts_on_edge(dim,0);

    // Store the number of points on an edge for quick reference
    for(i=0; i<dim; i++)
        pts_on_edge[i] = domain.getVarPoints();

    // Augment the number pieces on edges until done
    bool done = false;
    while(!done) {
        // Algorithm done if augmentations for all dimensions are maxed out.
        done = true;
        for(i=0; i<dim; i++)
            done = done && pcs_maxed[i]

        // Find the dimension most worthy of further augmentation
        if(!done) {
            int augment_dim;
            double biggest = 0;
            for(d=0; d<dim; d++) {
                if(!pcs_maxed[d]) {
                    double ratio = (pts_on_edge[d] / pcs_on_edge[d]);
                    if(ratio > biggest) {  
```
```c
    biggest = ratio;
    augment_dim = d;
}
}
}

// Augment the pieces_on_edge for the chosen dimension
pcs_on_edge[augment_dim]++;

// Calculate hypothetical number of boxes given new augmentation.
double hypothetical_total = 1;
for(d=0; d<dim; d++)
    hypothetical_total *= pcs_on_edge[d];

// If max_amount exceeded, undo the augment, and max-out the dimension
if(hypothetical_total > max_count) {
    pcs_maxed[ix] = true;
    pcs_on_edge[augment_dim]--;
}

// Cant have more pieces on an edge than points on an edge
if(pcs_on_edge[augment_dim] >= pts_on_edge[augment_dim])
    pcs_maxed[augment_dim] = true;
}

// Now build the uniform piece based on pts_on_edge and pcs_on_edge
IvPBox uniform_piece(dim);
for(d=0; d<dim; d++) {
    double edge_size = ceil(pts_on_edge[d] / pcs_on_edge[d]);
    uniform_piece.setPTS(d, 0, edge_size-1);
}
return(uniform_piece);
```

The heuristic progresses by growing the number of “pieces on an edge”, pcs_on_edge, on each dimension. The algorithm proceeds to grow the pcs_on_edge for each dimension until it cannot grow further. For example, in Figure 104 there are seven pieces on the x edge and five pieces on the y edge. The algorithm is initiated with a single piece on each edge, i.e., dimension, (line 4 in Listing 1). A Boolean is associated with each dimension indicating whether growth in that dimension has been maxed out. This vector is initiated on line 5. A dimension becomes maxed out if additional growth in that dimension means the requested piece count is exceeded (checked for in lines 37-48), or if the number of pieces on an edge is equal to the number of points on and edge of the IvP domain (checked for in lines 49-51). At each chance to grow the size of the uniform piece the most appropriate dimension is identified for growth (lines 22-33) by choosing the dimension with the largest ration of points on the edge to pieces on the edge (line 26).

Some examples of the heuristic are shown in Table 61. The domain shown in table has 1000 discrete choices for both the x and y variables. Given that the domain itself has an aspect ratio of one, not surprisingly, the generated uniform pieces also have roughly an aspect ratio of 1.0, and the number of pieces on each edge of the domain are also nearly equivalent.
Consider how the heuristic performs instead on the 3D domain shown in Table 62. The number of choices for the z variable is a tenth of that for the x and y variables. The results provided by the heuristic may or may not be the right overall, depending on the underlying function and application. In particular consider that when requesting 100 or 200 pieces, the z component of the resulting uniform piece is the entire z domain, i.e., there is only one piece on the z domain edge.

This heuristic has served fairly well in practice, but in cases where the user has insight into a better choice for the size and shape of the uniform piece, this can be overridden as discussed next.

### 39.2.3 Specifying the Uniform Piece Shape Explicitly

The piece shape used in a uniform IvP function can be set explicitly using the `uniform_piece` parameter in the Reflector `setParam()` function first mentioned in Section 38.4. For example, the uniform piece shown in Figure 106 can be requested as follows:

```java
reflector.setParam("uniform_piece", "discrete @ x:6,y:4");
int amt = reflector.create();
```

Compared to generating a uniform function by a simple piece-count request, the above two lines would replace the single line with the `create()` invocation, as in line 15 in Listing 6.
Figure 106: An IvP function made from an explicit piece shape request: An IvP domain is rendered over the two variables x, with 31 elements, and y with 21 elements. Requesting a uniform piece of size 6x4 would result in the rendered configuration. This piece can be specified with "discrete @ x:6,y:4" or "native @ x:6,y:20". This piece shape would not be resulting piece shape had the user simply requested 36 pieces given this domain.

In summary, when the Reflector create() function is called, the reflector-script begins and needs to know the size and shape of the piece used for uniform function creation. It may get this information by either explicitly configuring the piece shape, or implicitly by requesting a total number of pieces (as an argument to the create() function). If both requests are inadvertently invoked, the latter type of request is ignored and the explicit piece shape configuration is honored. If neither specification of piece shape is provided, a function with a single piece will be created (but perhaps further refined in later parts of the reflector-script). Use of the explicit piece shape request may be the preferred method for example if a domain includes a variable for vehicle heading and a uniform function is desired with pieces split on every three degrees, regardless of whether the domain contains 180, 360, or 720 choices for heading.

39.3 Optional Feature #2: IvP Functions with Directed Refinement

The directed-refinement feature of the Reflector is potentially useful when (a) the underlying function has distinct sub-regions that are harder to accurately represent with a piecewise linear approximation, and (b) when the user has insight into the location of those sub-regions. Use of the tool involves specifying both the region to direct further refinement, and the size of the piece to use in the refinement region. This is done using the uniform_piece, refine_region, and refine_piece parameters in the Reflector setParam() function first mentioned in Section 38.4. For example, the IvP function shown in Figure 105 would be generated with the following lines:
Listing 39.2: An example configuration of the Reflector tool using directed refinement.

```java
reflector.setParam("uniform_piece", "discrete @ x:5,y:5");
reflector.setParam("refine_region", "native @ x:10:24,y:-25:20");
reflector.setParam("refine_piece", "discrete @ x:2,y:2");
reflector.create();
```

When the `create()` function is invoked in the last line above, the reflector-script will involve two of the components of the reflector-script mentioned in Section 39.1.1. The first line configures the initial uniform function phase, and the middle two lines configure the directed refinement phase by declaring a sub-region (second line) and a uniform piece to be applied to that sub-region (third line). Multiple directed refinements can be configured and queued for inclusion in the reflector-script by adding further `refine_region - refine_piece` pairs prior to the invocation of the `create()` function. They must be added in pairs however since the `refine_piece` is always associated with the last specified `refine_region`.

For an illustrative case we return to the Gaussian function rendered in Figure 103:

\[
f(x, y) = A e^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})},
\]

where \( A = 150, x_0 = 50, y_0 = -150 \) and \( \sigma = 32.4 \). This function apparently has a sub-region of the domain where the function is very nonlinear and otherwise quite linear outside the sub-region. The use of directed-refinement begins by building an initial uniform function as shown in Figure 107, conceding for now that the approximation will be poor in the sub-region around the peak.

![Figure 107: An initial IvP function approximation](image)

The `Reflector` first creates an initial simple uniform function with fairly large pieces, conceding for the time being poor performance in approximating the underlying function in areas near the peak of the underlying function.
The initial uniform function was created by requesting 50 pieces, and a function with 49 pieces was subsequently generated. The sub-region shown in Figure 108 was identified for directed refinement, with much smaller pieces used in the sub-region.

![An IvP function generated with directed-refinement](image)

Figure 108: **An IvP function generated with directed-refinement**: After an initial uniform function has been generated, the Reflector refines the function on the prescribed sub-domain of the function with much smaller pieces.

The results in Table 63 below were generated by configuring the reflector-script to include directed-refinement on the underlying Gaussian function shown in Figure 108, in a manner similar to the four lines in Listing 2. Each row in the table below differs only in the size of the `refine_piece` shown in the second column.
<table>
<thead>
<tr>
<th>Case</th>
<th>Refine Edge Size</th>
<th>Total Pieces</th>
<th>Worst Error</th>
<th>Average Error</th>
<th>Time milliseecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>2607</td>
<td>0.7760</td>
<td>0.0104</td>
<td>38.9</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>1687</td>
<td>0.7760</td>
<td>0.0123</td>
<td>25.6</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>1162</td>
<td>0.7760</td>
<td>0.0149</td>
<td>17.8</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>847</td>
<td>0.7760</td>
<td>0.0178</td>
<td>13.0</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>682</td>
<td>0.9093</td>
<td>0.0214</td>
<td>10.2</td>
</tr>
<tr>
<td>F</td>
<td>9</td>
<td>535</td>
<td>1.1895</td>
<td>0.0254</td>
<td>8.2</td>
</tr>
<tr>
<td>G</td>
<td>10</td>
<td>447</td>
<td>1.4589</td>
<td>0.0302</td>
<td>6.7</td>
</tr>
<tr>
<td>H</td>
<td>11</td>
<td>367</td>
<td>1.8263</td>
<td>0.0351</td>
<td>5.7</td>
</tr>
<tr>
<td>I</td>
<td>12</td>
<td>295</td>
<td>2.2470</td>
<td>0.0409</td>
<td>4.7</td>
</tr>
<tr>
<td>J</td>
<td>13</td>
<td>262</td>
<td>2.6527</td>
<td>0.0472</td>
<td>4.1</td>
</tr>
<tr>
<td>K</td>
<td>14</td>
<td>231</td>
<td>3.0321</td>
<td>0.0540</td>
<td>3.6</td>
</tr>
<tr>
<td>L</td>
<td>15</td>
<td>202</td>
<td>3.5886</td>
<td>0.0615</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Table 63: Results of directed-refinement: Characteristics of 12 different IvP functions approximating the underlying function shown in Figure 107 and 108. Each function is built by starting with an initial uniform function and then performing directed refinement over the region \(-50 \leq x \leq 150\) by \(-250 \leq y \leq -50\). The refinement piece size shown in the second column is the parameter that results in the 12 different functions.

For the observed errors reported in columns 4 and 5, the domain was sampled for each resulting IvP function at 50,000 random points for comparison between the value provided by the IvP function against the underlying function. The average time to create the IvP function, noted in the last column, was taken by averaging 100 creations since the precision of the timer used was 100 milliseconds. The data shown here are meant to show the relationship between parameters, not necessarily an indication of how fast things run on “typical” platforms. That being said, this data is from a Dell laptop containing a Pentium chip with about 2.0 GHz processor, with a codebase compiled without typical gcc optimization options.

The trends in the table are as one would expect. As the number of pieces is decreased, the average error and worst error increase, and the time to create the IvP function is decreased. The question is whether this technique offers the ability to improve in all three metrics, piece-count, function accuracy, and creation time, simultaneously compared to using a simple uniform function without directed refinement. The answer is yes. Evidence can be seen of this by comparing Table 63 with Table 60. We look for cases in Table 63 that dominate cases in Table 60. A case that dominates another is stronger or equal in all three performance metrics simultaneously. Case (a) dominates case (4). Case (b) dominates cases (5),(4). Case (c) dominates cases (6),(5). Case (d) dominates cases (6),(5). Case (e) dominates cases (7),(6).

39.4 Optional Feature #3: IvP Functions with Smart Refinement

39.4.1 Potential Advantages

The smart-refinement algorithm works by further refining an existing IvP function based on an (automated) estimate of which pieces need refinement the most. There are two key ideas in this algorithm. First, no insight into the underlying function form is required by the user, unlike the directed-refinement tool. Second, the prioritization of pieces is based on the apparent fit between a piece’s linear function and the underlying function, for the sub-domain of that piece. This
determination of fit can be measured by performing very little extra computations beyond the already required calculations performed during linear regression for each piece during the uniform and directed-refinement phases. In short, there is typically very little reason not to invoke this tool to some degree.

39.4.2 The Smart-Refinement Algorithm

The smart-refinement algorithm utilizes information collected during the creation of pieces earlier in the reflector-script, during the initial uniform function phase which is always invoked, and directed-refinement phase which is optionally invoked. During these phases, pieces are formed and linear regression is performed to determine the linear function associated with each piece.

To perform linear regression for a new piece, the underlying function over \( k \) variables is sampled at \( n = 2k + 1 \) points (the corners and the middle point) to produce \( f(x_1) \ldots f(x_n) \), and these \( n \) values are used to determine the linear function for that piece:

\[
    f'(x) = c_1 x_1 + \ldots + c_k x_k + b.
\]

The same \( n \) points are again evaluated using this newly determined linear function instead, producing another set of \( n \) values \( f'(x_1) \ldots f'(x_n) \). The regression score is determined by:

\[
    \text{regression score} = \sqrt{\sum_{i=1}^{n} (f'(x_i) - f(x_i))^2}
\]

The regression score is then inserted into a priority queue along with a reference to the piece that generated the score. The idea is shown in Figure 109. This algorithm for implementing a priority queue can be found in [40]. The priority queue implemented in the IvPBuild Toolbox is modified slightly to be a fixed-length queue. Insertion and retrieval time is \( O(n \log(n)) \).
The **Reflector** instance maintains this priority queue only if smart-refinement is activated. The pieces made during the initial uniform function and directed-refinement parts of the reflector-script are stored in the priority queue. The smart-refinement proceeds by repeatedly popping the top priority piece from the queue for further refinement. By further refinement of a piece, we mean splitting a piece and replacing the piece with the two new pieces after performing regression on the two new pieces. The piece is split along the dimension with the largest edge. These two new pieces are then also inserted in the priority queue for possible further refinement.

An example of the smart-refinement algorithm applied to the same Gaussian function shown in Figure 108 is shown below in Figure 110.
The results in Table 64 show the results of applying the smart-refinement algorithm to the function in Figure 102. Each row in the table shows the results from creating first a pure uniform function, and then a further refined function using an additional 75% more pieces with smart-refinement. The left-hand side of the table is the same as Table 60, duplicated here for ease of comparison. Compare for example the smart-refine function with 175 pieces in Case 10 against the pure uniform function with 400 pieces in Case 7. The former not only has less pieces, but is more accurate and took less time to create. It dominates the pure uniform function, i.e., is simultaneously better in all measures of performance. This is similar to the way directed-refinement dominates pure uniform functions, but in the case of smart-refinement, no insight into the underlying function form was required!
<table>
<thead>
<tr>
<th>Case</th>
<th>Edge Size</th>
<th>Pieces</th>
<th>Worst Error</th>
<th>Avg Error</th>
<th>Time msec</th>
<th>Pieces</th>
<th>Worst Error</th>
<th>Avg Error</th>
<th>Time msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>2500</td>
<td>1.4589</td>
<td>0.0232</td>
<td>39.9</td>
<td>3445</td>
<td>0.8512</td>
<td>0.0139</td>
<td>70.1</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>1156</td>
<td>3.4532</td>
<td>0.0551</td>
<td>18.9</td>
<td>2023</td>
<td>1.3241</td>
<td>0.0224</td>
<td>47.3</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>625</td>
<td>5.5855</td>
<td>0.1014</td>
<td>10.4</td>
<td>1093</td>
<td>1.9834</td>
<td>0.0297</td>
<td>27.6</td>
</tr>
<tr>
<td>7</td>
<td>25</td>
<td>400</td>
<td>7.79764</td>
<td>0.1585</td>
<td>6.5</td>
<td>700</td>
<td>2.5924</td>
<td>0.0362</td>
<td>18.1</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>289</td>
<td>12.0347</td>
<td>0.2303</td>
<td>4.7</td>
<td>505</td>
<td>2.3905</td>
<td>0.0480</td>
<td>11.3</td>
</tr>
<tr>
<td>9</td>
<td>40</td>
<td>169</td>
<td>24.2977</td>
<td>0.3919</td>
<td>2.8</td>
<td>295</td>
<td>12.6192</td>
<td>0.0885</td>
<td>7.5</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>100</td>
<td>18.2113</td>
<td>0.5917</td>
<td>1.6</td>
<td>175</td>
<td>3.4166</td>
<td>0.1194</td>
<td>4.3</td>
</tr>
<tr>
<td>11</td>
<td>75</td>
<td>49</td>
<td>42.0652</td>
<td>1.2143</td>
<td>0.9</td>
<td>85</td>
<td>7.9236</td>
<td>0.3100</td>
<td>2.3</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>25</td>
<td>30.3938</td>
<td>2.0285</td>
<td>0.5</td>
<td>43</td>
<td>12.3887</td>
<td>0.5188</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 64: In each case an initial uniform function was created with the number of pieces indicated in column 3. The qualities of the function in terms of accuracy and time are shown in columns 4-6. Each function was then augmented with 75% additional pieces using the smart-refinement algorithm with the resulting qualities shown in columns 7-10.

The smart-refine algorithm is limited by the degree to which the regression score is accurate for each piece entered into the queue. Note for example, Case 9 in Table 64, where the worst error detected for the smart-refine function is anomalous and significantly higher than that noted in functions with far fewer pieces. The error of 12.6192 occurred in some piece that apparently did not report a high regression score. This is likely due to an unfortunate case where the points sampled for use in generating the linear function all fit the resulting linear function very well, but the non-sampled points did not fit well. The idea is shown in Figure 111.

![Figure 111: Regression scoring gone awry](image)

Figure 111: **Regression scoring gone awry**: Assessing the regression score can be misleading in cases where the derived linear function fits each sampled point very well but otherwise poorly fits the underlying function.

### 39.4.3 Invoking the Smart-Refine Algorithm in the Reflector

The smart-refine tool can be included in the refine-script in a few different ways. The first is to simply indicate how many pieces to use during the smart-refine process. The number of pieces is addition to any pieces that may already be present after the initial uniform function has bee created and after any directed-refinement has been performed.
reflector.setParam("smart_amount", 400);

Alternatively the number of pieces to be used in smart-refine can be given in terms of the percentage of additional pieces beyond what has already been created at the time of invocation of the smart-refine part of the refine-script. The argument is a non-negative integer value:

reflector.setParam("smart_percent", 35);

There is a third parameter smart_thresh that can affect how many pieces are used in the smart-refine phase. The value passed with this parameter is a regression score such that if the current top element of the priority queue has a score below this threshold, smart-refinement will terminate early, before the target piece amount specified by smart_amount or smart_percent has been reached:

reflector.setParam("smart_thresh", 0.05);

Regression scores represent the raw discrepancy between the underlying function and the linear approximation, and in general do not reflect any normalization. For example, depending on the function, the value of 0.05 above could be a relatively large value resulting in an early termination of refinement, or a relatively small threshold that cannot be met without thousands of additional pieces. The user of this parameter needs to have some knowledge of the range of the underlying function.

Finally, the simplest way of invoking the smart-refinement tool is by specifying the number of pieces as the second argument in the create() function call, and the smart threshold as the third argument:

reflector.create(1000, 400, 0.5);

The above will result in an initial uniform function with 1000 pieces and an additional 400 pieces used for smart-refinement. The full additional 400 pieces will be generated only if the threshold is not reached along the way. The above is equivalent to:

reflector.setParam("uniform_amount", 1000)
reflector.setParam("smart_amount", 400)
reflector.setParam("smart_thresh", 0.5)
reflector.create();

Accepting these three common parameters as arguments to the create() function call is simply for convenience. If provided, they override the setting from prior calls to setParam().

39.5 Optional Feature #4: IvP Functions with Auto-Peak Refinement

39.5.1 Potential Advantages

The auto-peak algorithm is the last optional algorithm in the reflector-script. The objective is also to build a more accurate IvP function representing the underlying function. The metric of accuracy referenced up to this point has been the average error and worst error observed from a number of random sample points. For example Table 64 reported error in this way. Another metric of accuracy is the degree to which the maximum peak of the function, represented by a point in the discrete
IvP domain, agree between the underlying function and the IvP function. For example, if the peak of the underlying function is "heading=133, speed=3.2" and the peak of the IvP function is "heading=129, speed=3.0" the IvP function could still be rated well in terms of error metrics from sampling the entire domain. However, the peak of the function is probably the most important part of the underlying function to represent precisely. When the IvP function happens to be the only function or dominating function influencing the vehicle at that moment, the peak of the function is the output of the solver. The auto-peak refinement focuses on this aspect of the function, without user insight into where the actual peak occurs in the underlying function.

### 39.5.2 The Auto-Peak Algorithm

The auto-peak algorithm proceeds by repeatedly refining the single piece in the IvP function that is believed to contain the maximum peak until that one piece contains only a single point in the IvP domain. It takes advantage of the fact that for a given piece in an IvP function, its maximum value, over the piece interval and linear interior function, can be rapidly calculated. The basic algorithm steps are as follows:

Listing 39.3: An overview of the auto-peak algorithm.

1. Populates the priority queue with the max-value for each piece.
2. If the top piece in the priority queue has only one IvP domain point, go to 10.
3. If the number of pieces added so far has reached an upper limit, go to 10.
4. Pop the top piece in the priority queue for further refinement.
5. Split the piece along the longest edge.
6. Build the new linear function for both pieces, noting max values.
7. Add the two new pieces back to the IvP function.
8. Add the two max-values back to the priority queue.
9. Go to Step 2.
10. Done.

The first step in the algorithm is to build a priority queue similar to the priority queue used in the smart-refinement algorithm. In this case, the score associate with each piece in the queue is the maximum value for the given piece, as depicted in Figure 112 below. The maximum value is calculated quickly and directly from the coefficients of the linear function associated with the piece. When the auto-peak algorithm is initiated, it works with the IvP function generated thus far during the prior phases of the reflector-script. The priority queue is built by evaluating the max-value for all pieces in this function.
Figure 112: Priority queue keyed with maximum utility scores: The Reflector uses a balanced priority queue based on a max utility score to determine which pieces could benefit the most from further refinement. Each node in the tree keeps a pointer to the piece that generated the maximum utility key.

The algorithm terminates when either the top piece in the priority queue contains only a single IvP domain point, or when auto-peak refinement has generated a total of new pieces that exceeds a specified optional limit. This is checked in steps 2-3 in Listing 3. When a piece is selected for further refinement, it is split along the longest edge creating two pieces (one new piece) regardless of the number of edges or dimensions. Linear regression is performed on the two pieces and they are added back to the IvP function and the priority queue. Typically, but not necessarily, the piece with the maximum value in the priority queue is a result of the most recent refinement.

39.5.3 Invoking the Auto-Peak Algorithm in the Reflector

Use of the auto-peak tool is done using the auto_peak and auto_peak_max_pcs parameters in the Reflector setParam() function first mentioned in Section 38.4. The auto_peak parameter simply turns the tool on or off, with true or false. The auto_peak_max_pcs parameter sets an upper limit on the number of new pieces introduced to an IvP function during the auto-peak phase. The following shows an example usage:

```java
reflector.setParam("auto_peak", "true")
reflector.setParam("auto_peak_max_pcs", 100)
reflector.create(1000);
```

The upper limit on pieces is typically not needed, and the default value is no limit. The algorithm tends to reach termination quickly because the piece with the maximum point tends to always be at the top of the priority queue, and in cases where the top ranked piece does not contain the maximum point, this is resolved quickly as the piece is split. Nevertheless, this upper limit is available for the conservative user. It is worth noting that, for underlying functions where the maximum value is part of a large plateau, the auto-peak tool is likely to have little benefit.
40 An Implementation Example - the SimpleWaypoint Behavior

In this section an example IvP behavior is presented. It is a simplified waypoint behavior version of the waypoint behavior in the standard suite of behaviors distributed with the MOOS-IvP public software bundle. The class name for this behavior is BHV_SimpleWaypoint. This behavior is distributed in the moos-ivp-extend repository and should build out of the box. After going through the class itself, later in this section example missions, also distributed with moos-ivp-extend, for running the behavior are discussed.

40.1 The SimpleWaypoint Behavior Class Definition

The SimpleWaypoint behavior is configured with four parameters: a single waypoint given in terms of local \( x \) and \( y \) coordinates, a transit speed in meters per second, and a radius in meters around the destination point within which the vehicle will be declared to have arrived at its waypoint. The behavior, at every iteration of the helm loop, notes the vehicle’s own position in \( x \) and \( y \) local coordinates. The idea is shown in Figure 113.

![Figure 113: The SimpleWaypoint behavior](image)

Figure 113: The SimpleWaypoint behavior: The SimpleWaypoint behavior works with a single waypoint. The location of the waypoint is stored in the local variable \( m_{\text{nextpt}} \) and is set during behavior configuration. The local variables \( m_{\text{osx}} \) and \( m_{\text{osy}} \) reflect the current vehicle (ownship) position updated at every helm iteration. The \( m_{\text{arrival\_radius}} \) determines how close the vehicle needs to be from the waypoint destination before declaring completion.

The BHV_SimpleWaypoint class definition is given below in Listing 1. Note that it is declared to be a subclass of the IvPBehavior superclass on line 8. The three helm-invoked overloadable functions are declared on lines 13-15. The constructor is defined to take an IvPDomain as an argument. The helm will instantiate each behavior with the same helm-configured domain as an argument to a behavior constructor.

Listing 40.1: BHV_SimpleWaypoint.h - the class definition for the "simple waypoint" behavior.

```cpp
1 #ifndef BHV_SIMPLE_WAYPOINT_HEADER
2 #define BHV_SIMPLE_WAYPOINT_HEADER
3
4 #include <string>
5 #include "IvPBehavior.h"
6 #include "XYPoint.h"
7
8 class BHV_SimpleWaypoint : public IvPBehavior {
```
The two configuration parameters depicted in Figure 113, the waypoint and arrival radius, are declared on lines 23-124. The two remaining configuration parameters, "speed" and "ipf_type" are on the following two lines. The former sets the ideal speed for waypoint transiting, and the latter indicates the type of IvP function to be generated. Two different ways of generating an IvP function are implemented in this behavior to demonstrate two different tools. The last part, lines 33-36 are the hooks needed for each behavior class to implement the dynamic loading of behaviors into the helm. These lines are therefore not present for behaviors compiled into the IvP helm. These lines are very pertinent to the discussion of extending the helm.

40.2 The SimpleWaypoint Behavior Class Implementation

The class implementation is given in Listings 2-7 below.

40.2.1 The SimpleWaypoint Behavior Constructor

The first part contains the class constructor in lines 17-34. On line 18, a call to the base-class constructor is made with the given domain. A default for the behavior name is also set on line 20. On line 21, the behavior declares that the domain over which it will produce an IvP function is comprised of both the course and speed variables. If the domain given to the behavior by the helm in the constructor does not have either of these variables, a null IvP domain will result in line 21. A null domain will make the behavior thereafter not capable of running, and is considered a fatal error, prompting the helm to post all-stop output values. This is purposely drastic. Configuring the behaviors in a vehicle mission where one of the behaviors is not runnable is worthy of stopping the helm and addressing the problem. Since this condition is checked for on all behaviors on each helm
iteration, this problem would always reveal itself at launch time, never during a mission, regardless of any dynamic behavior configurations during a mission.

On lines 25-27, default values for class member variables representing key behavior parameters are set in the constructor. In lines 30-31, class member variables representing behavior state variables are initialized. The grouping of member variables into two sets, one that represent parameter configurations and the other that otherwise represent behavior state maintained during operation, is merely a convention that has provided clarity in practice.

Listing 40.2: BHV_SimpleWaypoint.cpp - The SimpleWaypoint Behavior Constructor.

```cpp
1 #include <cstdlib>
2 #include <math.h>
3 #include "BHV_SimpleWaypoint.h"
4 #include "MBUtils.h"
5 #include "AngleUtils.h"
6 #include "BuildUtils.h"
7 #include "ZAIC_PEAK.h"
8 #include "OF_Coupler.h"
9 #include "OF_Reflector.h"
10 #include "AOF_SimpleWaypoint.h"

12 using namespace std;

14 //-----------------------------------------------------------
15 // Procedure: Constructor
16
17 BHV_SimpleWaypoint::BHV_SimpleWaypoint(IvPDomain gdomain) :
18 IvPBehavior(gdomain)
19 {
20   IvPBehavior::setParam("name", "simple_waypoint");
21   m_domain = subDomain(m_domain, "course,speed");
22   // All distances are in meters, all speed in meters per second
23   // Default values for configuration parameters
24   m_desired_speed = 0;
25   m_arrival_radius = 10;
26   m_ipf_type = "zaic";
27
28   // Default values for behavior state variables
29   m_osx = 0;
30   m_osy = 0;
31
32   addInfoVars("NAV_X, NAV_Y");
34 }
```

Finally, on line 33, the behavior declares two variables, `NAV_X` and `NAV_Y`, representing vehicle ownership position. The IvP helm, containing this behavior, will need to register for these two variables on the behavior's behalf. This is the hook where the behavior tells the helm what it needs from the MOOSDB. It is from these two variables that the behavior will populate its variables `m_osx` and `m_osy` representing the current vehicle position.

40.2.2 The SimpleWaypoint Behavior setParam() Function

In Listing 3 below, a key over-loadable behavior function is implemented, the setParam() function, in lines 39-69. This function handles the configuration of the behavior for its five parameters, "ptx", "pty", "ptz", "plx", and "ply". This example illustrates how to set default values for configuration parameters, and how to initialize behavior state variables.
"pty", "speed", "radius", and "ipf_type". An example configuration for this behavior is given in Listing 8. Behavior parameters defined at the IvPBehavior superclass level, such as name, condition, endflag, etc., are handled in the setParam() function of the superclass. The helm, when it handles a behavior parameter from a *.bhv file, first attempts to handle the parameter at the superclass level. If the IvPBehavior::setParam() function returns false, the helm passes the parameter-value pair to the behavior’s locally implemented version of setParam().

Listing 40.3: BHV_SimpleWaypoint.cpp - The setParam() function.

```cpp
36  /**************************************************************************
37  // Procedure: setParam - handle behavior configuration parameters
38  **************************************************************************/
39  bool BHV_SimpleWaypoint::setParam(string param, string val)
40  {
41     // Convert the parameter to lower case for more general matching
42     param = tolower(param);
43
44     double double_val = atof(val.c_str());
45     if((param == "ptx") && (isNumber(val))) {
46         m_nextpt.set_vx(double_val);
47         return(true);
48     }
49     else if((param == "pty") && (isNumber(val))) {
50         m_nextpt.set_vy(double_val);
51         return(true);
52     }
53     else if((param == "speed") && (double_val > 0) && (isNumber(val))) {
54         m_desired_speed = double_val;
55         return(true);
56     }
57     else if((param == "radius") && (double_val > 0) && (isNumber(val))) {
58         m_arrival_radius = double_val;
59         return(true);
60     }
61     else if(param == "ipf_type") {
62         val = tolower(val);
63         if((val == "zaic") || (val == "reflector") ||
64            (val == "ipf_type") {
65             m_ipf_type = val;
66             return(true);
67         }
68         return(false);
69     }
70     return(false);
```

A fair amount of error checking is done for parameter. For example, in setting the "speed" parameter, the string value is checked to ensure that is both numerical and larger than zero. Solid error checking implemented in this function is a very good idea that will save headaches down the road. This function should only return true if it has been passed a proper parameter-value pair. Another common practice is to perform a case insensitive parameter match, e.g., "pty" and "PTY" are both allowable configurations. This is done by converting the string representing the parameter to lower case in line 42. In this case, the tolower() function is defined in a local utility toolbox.
40.2.3 The SimpleWaypoint onIdleState() and postViewPoint() Functions

The onIdleState() function, lines 74-77, is only executed when the behavior is in the idle state, i.e., not in the running state. See Sections 7.4 and 7.6 for more on behavior states. In this behavior, the only task executed in the onIdleState() function is to publish a waypoint marker in the form of the MOOS variable VIEW_POINT.

Listing 40.4: BHV_SimpleWaypoint.cpp - The onIdleState() and postViewPoint() functions.

```cpp
71 //********************************************************************************
72 // Procedure: onIdleState
73 //********************************************************************************
74 void BHV_SimpleWaypoint::onIdleState()
75 {
76   postViewPoint(false);
77 }
78
79 //********************************************************************************
80 // Procedure: postViewPoint
81 //********************************************************************************
82 void BHV_SimpleWaypoint::postViewPoint(bool viewable)
83 {
84   m_nextpt.set_label(m_us_name + "'s next waypoint");
85   m_nextpt.set_type("waypoint");
86   m_nextpt.set_source(m_descriptor);
87   string point_spec;
88   if(viewable)
89     point_spec = m_nextpt.get_spec("active=true");
90   else
91     point_spec = m_nextpt.get_spec("active=false");
92   postMessage("VIEW_POINT", point_spec);
93 }
94
An example produced by this would be:

```
VIEW_POINT = "active,false:label,alder's next waypoint:
              type,waypoint:source,waypt_return:0,0,0"
```

In this case, due to the "active,false" component, the posting of this variable would serve to "erase" similar postings to this variable made in the onRunState() function described next. For more on how VIEW_POINT is consumed, see the documentation on the pMarineViewer application in [30].

40.2.4 The SimpleWaypoint Behavior onRunState() Function

Implementation of the onRunState() function is where the primary unique operation of the behavior is implemented. For the SimpleWaypoint behavior, the full function is in Listing 5 below. It is implemented in four parts:

- Part 1: Get the vehicle position from the information buffer.
- Part 2: Determine if the waypoint has been reached and possibly enter complete mode.
- Part 3: Build a status message regarding the waypoint for third party viewers.
Part 4: Build an IvP function with either the ZAIC or Reflector tool.

In the first part, lines 101-109, information from the information buffer is retrieved regarding the vehicle’s own position. This is done with the `getBufferDoubleVal()` function described in Section 7.5.1. In this behavior the result of the query to the buffer is stored in the `ok1` and `ok2` variables and subsequently checked and handled in lines 106-109. In this behavior, if essential information like the vehicle’s own position is missing, a warning is posted (line 107) and the `onRunState()` function returns without producing an objective function (line 108). In such a case the behavior would be considered to be in the running state, but not the active state for the present iteration.

A fair point to raise regarding Part 1 is the possibility that the vehicle’s position information is in the buffer but has become so old that it no longer reflects the vehicle’s true current position. In other words, what if the navigation module on board the vehicle has somehow shut down? First, in most situations with a vehicle implementing the backseat/front-seat driver architecture described in [35] and [34], a heartbeat monitor for the navigation system is typically put in place at the larger autonomy system level and an all-stop would be invoked overriding the helm. However, for the sake of having some fail-safe redundancy within the helm to handle this situation, the `no_starve` parameter could be used (Section 7.2.1) for this behavior, or any behavior since it is defined at the `IvPBehavior` superclass level. An example of it’s usage is shown in Listing 8, setting a `no_starve` threshold of 3 seconds for `NAV_X` and `NAV_Y`.

Listing 40.5: BHV_SimpleWaypoint.cpp - the `onRunState()` implementation.

```c++
96   //-------------------------------------------------------------
97   // Procedure: onRunState
98   // IvPFunction *BHV_SimpleWaypoint::onRunState()
99   {
100      // Part 1: Get vehicle position from InfoBuffer and post a
101         // warning if problem is encountered
102      bool ok1, ok2;
103      m_osx = getBufferDoubleVal("NAV_X", ok1);
104      m_osy = getBufferDoubleVal("NAV_Y", ok2);
105      if(!ok1 || !ok2) {
106          postWMessage("No ownship X/Y info in info_buffer.");
107          return(0);
108      }
109      
110      // Part 2: Determine if the vehicle has reached the destination
111         // point and if so, declare completion.
112      double dist = hypot((m_nextpt.x()-m_osx), (m_nextpt.y()-m_osy));
113      if(dist <= m_arrival_radius) {
114          setComplete();
115          postViewPoint(false);
116          return(0);
117      }
118      
119      // Part 3: Post the waypoint as a string for consumption by
120         // a viewer application.
121      postViewPoint(true);
122      
123      // Part 4: Build the IvP function with either the ZAIC tool
124         // or the Reflector tool.
125      IvPFunction *ipf = 0;
126      if(m_ipf_type == "zaic")
```
In Part 2 of the `onRunState()` function, in lines 111-118 of Listing 5, the determination of waypoint arrival is made. This is just a simple comparison between the current distance of the vehicle and the waypoint to the configured arrival radius in the parameter `m.arrival_radius`. If a determination of arrival is made, the behavior calls the `setComplete()` function. This function is defined at the behavior superclass level and was described in detail in Section 7.5.1. The invocation of this function will put the behavior in the group of `completed` behaviors on the next helm iteration. In the current iteration this behavior would be considered in the `running` state, but not the `active` state since the `onRunState()` function returns (line 117) without generating an objective function. See Section 6.5.3 for more on behavior run states.

In Part 3, the behavior generates a visual artifact for consumption by a viewer, for rendering the waypoint the behavior is using as its destination. This point is shown in Figures 118 and 119 with the label "Alder’s next waypoint". An example produced by this would be:

```
VIEW_POINT = "label=alder's next waypoint,type=waypoint,source=transit,x=60,y=-40"
```

Compare this to the value for `VIEW_POINT` generated in the `onIdleState()` function described in Section 40.2.3. This variable-value pair is generated by the behavior for posting on each invocation of the `onRunState()` even though the value posted does not generally change between iterations. The posting of the variable-value pair is done with the `postMessage()` function, described in Section 7.5.1. The invocation of `postMessage()` will result in an actual post to the MOOSDB only if the value of string posted changes. Successive duplicate postings are filtered out by the Duplication Filter. The Duplication Filter was described in Section 5.8 where alternative functions for overriding the filter are given to accommodate other situations. The `postMessage()` function was discussed in Section 7.5.1.

In Part 4 of the `onRunState()` function, in lines 124-135 of Listing 5, an IvP function is generated over a domain of heading and speed choices to reflect the goal of reaching a waypoint given a current vehicle position. For the purposes of providing an example usage of the IvPBuild Toolbox, the behavior is implemented to produced an IvP function using two different methods of the toolbox. The SimpleWaypoint behavior can be configured with the "ipf.type" parameter, as shown in Listing 3, to accept either the configuration of "zaic" or "reflector". In the example mission, mission "Alder" described below in Section 40.3, the helm is configured in Listing 8 to use the ZAIC tool on the outbound transit trip, and the Reflector tool on the return trip. The implementation of the `onRunState()` function merely checks the type of IvP function desired and makes the appropriate call to either the `buildFunctionWithZAIC()` function or the `buildFunctionWithReflector()` function. These two functions are described next.
40.2.5 The SimpleWaypoint Behavior buildFunctionWithZAIC() Function

When the SimpleWaypoint behavior is configured to generate an IvP function with the ZAIC tool, it invokes the function buildFunctionWithZAIC(), shown below in Listing 6. Of the three ZAIC tools described in Section 37, the ZAIC_PEAK is used in this behavior. It is used to generate two one-variable IvP functions. The first function is defined over the speed decision variable in lines 142-151. The second function is defined over heading decision variable. The functions are shown in Figure 114.

Listing 40.6: BHV_SimpleWaypoint.cpp - the buildFunctionWithZAIC() implementation.

```cpp
137 //-----------------------------------------------------------
138 // Procedure: buildFunctionWithZAIC
139 
140 IvPFunction *BHV_SimpleWaypoint::buildFunctionWithZAIC()
141 {
142 ZAIC_PEAK spd_zaic(m_domain, "speed");
143 spd_zaic.setSummit(m_desired_speed);
144 spd_zaic.setPeakWidth(0.5);
145 spd_zaic.setBaseWidth(1.0);
146 spd_zaic.setSummitDelta(0.8);
147 if(spd_zaic.stateOK() == false) {
148 string warnings = "Speed ZAIC problems " + spd_zaic.getWarnings();
149 postWMessage(warnings);
150 return(0);
151 }
152 
153 double rel_ang_to_wpt = relAng(m_osx, m_osy, m_nextpt.x(), m_nextpt.y());
154 ZAIC_PEAK crs_zaic(m_domain, "course");
155 crs_zaic.setSummit(rel_ang_to_wpt);
156 crs_zaic.setPeakWidth(0);
157 crs_zaic.setBaseWidth(180.0);
158 crs_zaic.setSummitDelta(0);
159 crs_zaic.setValueWrap(true);
160 if(crs_zaic.stateOK() == false) {
161 string warnings = "Course ZAIC problems " + crs_zaic.getWarnings();
162 postWMessage(warnings);
163 return(0);
164 }
165 
166 IvPFunction *spd_ipf = spd_zaic.extractIvPFunction();
167 IvPFunction *crs_ipf = crs_zaic.extractIvPFunction();
168 
169 OF_Coupler coupler;
170 IvPFunction *ivp_function = coupler.couple(crs_ipf, spd_ipf, 50, 50);
171 
172 return(ivp_function);
173 }
174 
```

The first step in creating an IvPFunction with the ZAIC_PEAK tool is to create an instance of the ZAIC_PEAK, on line 142, passing it the IvPDomain used by the behavior and set in the behavior constructor in Listing 2. The ZAIC constructor is also passed the name of the one variable in the IvPDomain for which to create the IvPFunction. The ZAIC_PEAK parameters, set in lines 143-146, are described in detail in Section 37.1.1. In lines 147-151, a check is made to determine whether the ZAIC_PEAK instance has been configured properly. The stateOK() function returns false if there were any configuration problems, and the string returned by getWarnings() function on line 148.
will provide insight into any configuration errors. The `postWMessage()` function on lines 149 and 162 will result in the helm posting to the MOOSDB the variable `BHV_WARNING` with the contents of string `warnings`. The `postWMessage()` function is discussed in Section 7.5.1 on page ??.

The second one-variable function, defined over `course`, is created with a second `ZAIC_PEAK` instance in lines 153-164. First, the angle between the current vehicle position and the waypoint destination is calculated on line 153, with a call to the `relAng()` function, defined in one of the MOOS-IvP utility libraries. The creation and configuration of the `ZAIC_PEAK` instance proceeds much in same way as for the first one. In this case, the `valuewrap` parameter is set to `true` on line 159 to indicate that the domain values should “wrap around”, that is, a course of 350 is 20 degrees separated from a course of 10 degrees, not separated by 340 degrees. The `valuewrap` parameter is discussed in Section 37.1.3 on page 295.

![Figure 114: IvP functions produced by the ZAIC_PEAK Tool](image)

*Figure 114: IvP functions produced by the ZAIC_PEAK Tool: The two functions produced by the Simple-Waypoint behavior by use of the ZAIC_PEAK tool would produce a function over `speed` with six pieces and a function over `heading` with three pieces.*

When these two functions are coupled using the `Coupler` tool, lines 169-170, an IvP function over the coupled decision 2D space is created as shown in Figure 115 below. The Coupler tool is discussed in Section 36.2.3 on page 291.
The two one-variable functions are combined with an equal weight of 50 on line 170. The choice of relative weight has a distinct influence over the resulting function. In Figure 116 below, two IvP functions similarly generated, but with alternative weights are shown. The Coupler, by default, normalizes the combined function to the range of $[0, 100]$. This can be turned off, or normalized with a different range as described in Section 36.2.3. The range of $[0, 100]$ is a common range for functions returned by an IvP behavior to the IvP Solver.

In each IvP function in Figures 115 and 116, the location of the peak of the function is the same. When this behavior is the only active behavior, the path taken by the vehicle will be the same regardless of the weights chosen to combine the two one-variable functions with the Coupler.
The only thing that matters is the value of the \textit{summit} parameters passed to the two ZAIC tools creating the two one-variable functions. The off-peak characteristics of the function begin to matter when the behavior is coordinated with functions from other behaviors. In the function on the right in Figure 116, two off-peak decisions with equal utility values are shown. One point represents a decision with the heading more toward the destination with a speed much higher than the desired speed, and the other decision represents a heading less toward the destination but with speed near the optimal speed. In the absence of mission metrics that clarify the relative utility of sub-optimal transit paths versus sub-optimal speeds, the construction of the off-peak shape of the objective function is typically a subjective decision of the behavior implementor.

40.2.6 The SimpleWaypoint Behavior buildFunctionWithReflector() Function

The Reflector tool can be used to generate objective functions that cannot otherwise be formed as the product of the coupling of two independent functions. This gives the behavior implementor more freedom to generate functions with off-peak characteristics more in-line with the goals of the behavior. The Reflector tool is described in detail in Sections 38 and 39.

The use of the Reflector tool in the SimpleWaypoint behavior is given in Listing 7 below. The Reflector generates an IvP function approximation of an underlying function, an instance of the \texttt{AOF\_SimpleWaypoint} class. The underlying function and the IvP function are defined over the same domain. This domain is passed to the underlying function in its constructor (line 183). The underlying function is passed required parameters (lines 184-188) and initialized (line 189). If any part of the initialization fails, a null IvP function is returned (line 180, 196).

\begin{verbatim}
Listing 40.7: BHV\_SimpleWaypoint.cpp - the buildFunctionWithReflector() implementation.

175  //-----------------------------------------------------------
176  // Procedure: buildFunctionWithReflector
177  IvPFunction *BHV_SimpleWaypoint::buildFunctionWithReflector() {
178     IvPFunction *ivp_function = 0;
179     bool ok = true;
180     AOF_SimpleWaypoint aof_wpt(m_domain);
181     ok = ok && aof_wpt.setParam("desired_speed", m_desired_speed);
182     ok = ok && aof_wpt.setParam("osx", m_osx);
183     ok = ok && aof_wpt.setParam("osy", m_osy);
184     ok = ok && aof_wpt.setParam("ptx", m_nextpt.x());
185     ok = ok && aof_wpt.setParam("pty", m_nextpt.y());
186     ok = ok && aof_wpt.initialize();
187     if(ok) {
188         OF_Reflector reflector(&aof_wpt);
189         reflector.create(1000);
190         ivp_function = reflector.extractIvPFunction();
191     }
192     return(ivp_function);
193 }
\end{verbatim}

The Reflector tool does its work (lines 191-193) after it has been determined that a proper instance of the underlying function, \texttt{AOF\_SimpleWaypoint}, has been created and initialized. The Reflector tool has several options for creating a piecewise defined IvP function, described later in

330
Sections 38 and 39. The simplest method is to specify the number of pieces desired in the piecewise function, in this case 1000 pieces were requested, on line 192. After creation, the IvP function is extracted from the Reflector (line 193), and returned (line 196).

An example of the IvP function created with the Reflector is shown in Figure 117. The function represents a preference for maneuvers that bring the vehicle toward the waypoint at a desired speed. The values of off-peak areas are evaluated based on (a) the rate of closure, (b) the rate of detour, and (c) the deviation from the desired speed. For this function generated by the Reflector, and the particular underlying function, it is not possible to generate a function of equal form using the ZAIC tools.

Figure 117: An IvP function created with the Reflector tool: The function represents a preference for maneuvers that bring the vehicle toward the waypoint at a desired speed. The values of off-peak areas are evaluated based on (a) the rate of closure, (b) the rate of detour, and (c) the deviation from the desired speed. Decisions of increasing speed are represented by points on the function radially farther from the center.

40.3 Running an Example Mission with the SimpleWaypoint Behavior

An example mission file, alder.moos, and behavior file, alder.bhv, have been configured to demonstrate the usage of the SimpleWaypoint behavior. The files may be found in the moos-ivp-extend tree. Refer back to Section 35.2 for information on obtaining this tree from the web. The example mission with the SimpleWaypoint behavior is called the Alder mission and is found by:

```bash
$ cd moos-ivp-extend/missions/alder
$ ls
$ README  alder.bhv  alder.moos
```

The behavior file is given in Listing 8 below. The SimpleWaypoint behavior is used twice. It is used to transit to a point and then to return to the starting point. The transiting use of the behavior is configured in lines 7-20, and the returning use of the behavior in lines 23-35. See [34] for further information regarding behavior files and their usage.

Listing 40.8: The alder.bhv file - For running the Alder example mission.
initialize DEPLOY = false
initialize RETURN = false

Behavior = BHV_SimpleWaypoint {
  name = transit_to_point
  pwt = 100
  condition = RETURN = false
  condition = DEPLOY = true
  endflag = RETURN = true
  speed = 2.0 // meters per second
  radius = 8.0
  ptx = 60
  pty = -40
  ipf_type = zaic
}

Behavior = BHV_SimpleWaypoint {
  name = waypt_return
  pwt = 100
  condition = RETURN = true
  condition = DEPLOY = true
  speed = 2.0
  radius = 8.0
  ptx = 0
  pty = 0
  ipf_type = reflector
}

Both behaviors are idle upon startup. Presumably the transit behavior is activated first by setting `DEPLOY=true`, and the second instance of the behavior is activated when the transit behavior completes and sets its endflag. The `alder.moos` file is not discussed here, but may be examined in the tree.

The example mission may be started by:

```
$ cd moos-ivp-extend/missions/alder
$pAntler alder.moos
```

The `pMarineViewer` window should launch, and look similar to image in Figure 118. After clicking on the `DEPLOY` button in the lower right corner, the transiting instance of the SimpleWaypoint behavior becomes active and the vehicle begins to form a track to the waypoint as shown. Clicking on the `DEPLOY` button initiates a MOOS poke on the MOOSDB connected to both the `pMarineViewer`, and `pHelmIvP` application. This mission setup is quite similar to the Alpha mission discussed in [34].
Figure 118: The SimpleWaypoint behavior in action: After the user clicks on the DEPLOY button, the condition for the transiting SimpleWaypoint behavior is satisfied (line 13 in Listing 8).

Figure 119: The SimpleWaypoint behavior in action: After the vehicle has reached the waypoint prescribed in the transiting instance of the SimpleWaypoint behavior, the second instance of the SimpleWaypoint behavior, returning to the start point, becomes active.
A  Use of Logic Expressions

Logic conditions are employed in both the pHelmIVP, uTimerScript, uQueryDB and alogcheck applications, to condition certain activities based on the prescribed logic state of elements of the MOOSDB. The use of logic conditions in the helm is done in behavior file (.bhv file). For the uTimerScript application, logic conditions are used in the configuration block of the mission file (.moos file). The MOOS application using logic conditions maintains a local buffer representing a snapshot of the MOOSDB for variables involved in the logic expressions. The key relationships and steps are shown in Figure 120:

![Figure 120: Logic conditions in a MOOS application](image)

Step 1: the application registers to the MOOSDB for any MOOS variables involved in the logic expressions. Step 2: The MOOS application reads incoming mail from the MOOSDB. Step 3: Any new mail results in an update to the information buffer. Step 4: Within the application's Iterate() method, the logic expressions are evaluated based on the contents of the information buffer.

The logic conditions are configured as follows:

```
condition = <logic-expression>
```

The parameter `condition` is case insensitive. When multiple conditions are specified, it is implied that the overall criteria for meeting conditions is the conjunction of all such conditions. In what remains below, the allowable syntax for `<logic-expression>` is described.

Simple Relational Expressions

Each logic expression is comprised of either Boolean operators (and, or, not) or relation operators (≤, <, ≥, >, =, ! =). All expressions have at least one relational expression, where the left-hand side of the expression is a MOOS variable, and the right-hand side is a literal (either a string or numerical value). The literals are treated as a string value if quoted, or if the value is non-numerical. Some examples:

```
condition = (DEPLOY = true)       // Example 1
condition = (QUALITY >= 75)       // Example 2
```

Variable names are case sensitive since MOOS variables in general are case sensitive. In matching string values of MOOS variables in Boolean conditions, the matching is case insensitive. If for example, in Example 1 above, the MOOS variable DEPLOY had the value "TRUE", this would satisfy the condition. But if the MOOS variable deploy (lowercase is unconventional, but legal) had the value "true", this would not satisfy Example 1.
Simple Logical Expressions with Two MOOS Variables

A relational expression generally involves a variable and a literal, and the form is simplified by insisting the variable is on the left and the literal on the right. A relational expression can also involve the comparison of two variables by surrounding the right-hand side with $(())$. For example:

```plaintext
condition = (REQUESTED_STATE != $(RUN_STATE))  // Example 3
```

The variable types need to match or the expression will evaluate to false regardless of the relation. The expression in Example 3 will evaluate to false if, for example, REQUESTED_STATE="run" and RUN_STATE=7, simply because they are of different type, and regardless of the relation being the inequality relation.

Complex Logic Expressions

Individual relational expressions can be combined with Boolean connectors into more complex expressions. Each component of a Boolean expression must be surrounded by a pair of parentheses. Some examples:

```plaintext
condition = (DEPLOY = true) or (QUALITY >= 75)  // Example 4
condition = (MSG != error) and !((K <= 10) or (w != 0))  // Example 5
```

A relational expression such as $(w != 0)$ above is false if the variable $w$ is undefined. In MOOS, this occurs if the variable has yet to be published with a value by any MOOS client connected to the MOOSDB. A relational expression is also false if the variable in the expression is the wrong type, compared to the literal. For example $(w != 0)$ in Example 5 would evaluate to false even if the variable $w$ had the string value "alpha" which is clearly not equal to zero.
## B Behavior Summaries

### Parameter Summary for BHV_Waypoint

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>capture_radius</td>
<td>double</td>
<td>7</td>
<td>-</td>
<td>0</td>
<td>121</td>
</tr>
<tr>
<td>cycleflag</td>
<td>MOOSVAR=value</td>
<td>CYCLED=true</td>
<td>yes</td>
<td>-</td>
<td>116</td>
</tr>
<tr>
<td>wptflag</td>
<td>MOOSVAR=value</td>
<td>MEASURE=true</td>
<td>yes</td>
<td>-</td>
<td>123</td>
</tr>
<tr>
<td>lead</td>
<td>double</td>
<td>10</td>
<td>-</td>
<td>-1</td>
<td>122</td>
</tr>
<tr>
<td>lead_damper</td>
<td>double</td>
<td>2</td>
<td>-</td>
<td>-1</td>
<td>122</td>
</tr>
<tr>
<td>lead_on_start</td>
<td>Boolean string</td>
<td>TRUE</td>
<td>no</td>
<td>false</td>
<td>122</td>
</tr>
<tr>
<td>slip_radius</td>
<td>double</td>
<td>18</td>
<td>-</td>
<td>0</td>
<td>121</td>
</tr>
<tr>
<td>order</td>
<td>string</td>
<td>reverse</td>
<td>no</td>
<td>normal</td>
<td>120</td>
</tr>
<tr>
<td>point</td>
<td>string</td>
<td>20,5</td>
<td>yes</td>
<td>-</td>
<td>120</td>
</tr>
<tr>
<td>points, polygon</td>
<td>string</td>
<td>0,0:45,0:45,80:0,0</td>
<td>yes</td>
<td>-</td>
<td>120</td>
</tr>
<tr>
<td>post_suffix</td>
<td>string</td>
<td>IKE</td>
<td>yes</td>
<td>**&quot;</td>
<td>116</td>
</tr>
<tr>
<td>repeat</td>
<td>int</td>
<td>3</td>
<td>no</td>
<td>0</td>
<td>120</td>
</tr>
<tr>
<td>speed</td>
<td>double</td>
<td>1.2</td>
<td>-</td>
<td>0</td>
<td>116</td>
</tr>
<tr>
<td>visual_hints</td>
<td>string</td>
<td>edge_size=1</td>
<td>no</td>
<td>-</td>
<td>126</td>
</tr>
</tbody>
</table>

Table 65: Parameters for the BHV_Waypoint behavior.

### Variables posted by the BHV_Waypoint Behavior

The following variables may be posted by the BHV_Waypoint behavior in addition to any configured flags (with the runflag, idleflag, activeflag, inactiveflag, and cycleflag parameters).

- **VIEW_POINT, VIEW_SEGLIST, WPT_INDEX, CYCLE_INDEX, WPT_STAT**

The variable names used may be changed with the `post_mapping` parameter. See page 103.

### Example Behavior File Configuration for BHV_Waypoint

*Listing B.1 - An example BHV_Waypoint configuration.*
Behavior = BHV_Waypoint
{
  name = waypt_survey
  priority = 100
  updates = WPT_SURVEY_UPDATES
  condition = (DEPLOY == true) or (SURVEY == on))
  endflag = SURVEY = COMPLETE
  points = label, survey_points=-57,-60:-70,-109:-77,-144:-51
  speed = 3.0 // meters per second
  capture_radius = 8.0 // meters
  nm_radius = 16.5 // meters
  repeat = 0 // number of iterations
  lead = 10 // meters
Parameter Summary for BHV_OpRegion

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>660</td>
<td>yes</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>yes</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>89</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>polygon</td>
<td>string</td>
<td>0,0:45,0:45,80:0,80:0,0</td>
<td>-</td>
<td>-</td>
<td>129</td>
</tr>
<tr>
<td>max_depth</td>
<td>double</td>
<td>200</td>
<td>-</td>
<td>0</td>
<td>130</td>
</tr>
<tr>
<td>min_altitude</td>
<td>double</td>
<td>25</td>
<td>-</td>
<td>0</td>
<td>130</td>
</tr>
<tr>
<td>max_time</td>
<td>double</td>
<td>3600</td>
<td>-</td>
<td>0</td>
<td>130</td>
</tr>
<tr>
<td>trigger_entry_time</td>
<td>double</td>
<td>1.5</td>
<td>-</td>
<td>0</td>
<td>129</td>
</tr>
<tr>
<td>trigger_exit_time</td>
<td>double</td>
<td>2.4</td>
<td>-</td>
<td>0</td>
<td>129</td>
</tr>
<tr>
<td>visual_hints</td>
<td>string</td>
<td>edge_size=1</td>
<td>no</td>
<td>-</td>
<td>127</td>
</tr>
</tbody>
</table>

Table 66: Parameters for the BHV_OpRegion behavior.

Example Behavior File Configuration for BHV_OpRegion

Listing B.2 - An example BHV_OpRegion configuration.

```plaintext
0 Behavior = BHV_OpRegion
1 {
2   name = bhv_opregion
3   polygon = label,opregion : -57,-60 : -70,-109 : -77,-144
4   max_depth = 50 // meters
5   min_altitude = 10 // meters
6   max_time = 3600 // seconds
7   trigger_entry_time = 0.5 // seconds
8   trigger_exit_time = 1.0 // seconds
9 }
```

338
Parameter Summary for BHV_Loiter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runiflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR, double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR, MOOSVAR</td>
<td>FOO, BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>acquire_dist</td>
<td>double</td>
<td>15</td>
<td>-</td>
<td>10</td>
<td>138</td>
</tr>
<tr>
<td>capture_radius</td>
<td>double</td>
<td>10</td>
<td>-</td>
<td>0</td>
<td>138</td>
</tr>
<tr>
<td>center_activate</td>
<td>Boolean</td>
<td>true</td>
<td>no</td>
<td>false</td>
<td>136</td>
</tr>
<tr>
<td>center_assign</td>
<td>string</td>
<td>present_position</td>
<td>no</td>
<td>-</td>
<td>136</td>
</tr>
<tr>
<td>clockwise</td>
<td>Boolean string</td>
<td>FALSE</td>
<td>no</td>
<td>true</td>
<td>138</td>
</tr>
<tr>
<td>polygon</td>
<td>string</td>
<td>0,0:45,0:45,80:0,80:0,0</td>
<td>yes</td>
<td>-</td>
<td>136</td>
</tr>
<tr>
<td>post_suffix</td>
<td>string</td>
<td>REGION-1</td>
<td>yes</td>
<td>&quot;&quot;</td>
<td>138</td>
</tr>
<tr>
<td>speed</td>
<td>double</td>
<td>1.5</td>
<td>-</td>
<td>0</td>
<td>137</td>
</tr>
<tr>
<td>slip_radius</td>
<td>double [0,100]</td>
<td>25</td>
<td>-</td>
<td>0</td>
<td>138</td>
</tr>
<tr>
<td>spiral_factor</td>
<td>double</td>
<td>45</td>
<td>-</td>
<td>99</td>
<td>138</td>
</tr>
<tr>
<td>visual_hints</td>
<td>string</td>
<td>edge_size=1</td>
<td>no</td>
<td>-</td>
<td>139</td>
</tr>
<tr>
<td>xcenter_assign</td>
<td>double</td>
<td>20</td>
<td>no</td>
<td>-</td>
<td>136</td>
</tr>
<tr>
<td>ycenter_assign</td>
<td>double</td>
<td>25</td>
<td>no</td>
<td>-</td>
<td>136</td>
</tr>
</tbody>
</table>

Table 67: Parameters for the BHV_Loiter behavior.

Example Behavior File Configuration for BHV_Loiter

Listing B.3 - An example BHV_Loiter configuration.

```plaintext
0  Behavior = BHV_Loiter
1  {
2    name = loiter_alpha
3    put = 100
4    duration = 3600 // One hour
5    updates = LOITER_ALPHA_UPDATES
6    polygon = radial:100,-100,80,12
7    speed = 3.0
8    capture_radius = 8.0
9    slip_radius = 16.0
10   clockwise = true
11   acquire_dist = 25
```
14   center_assign = present_position
14 }

340
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>yes</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR, double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>basewidth</td>
<td>double</td>
<td>0.5</td>
<td>-</td>
<td>0</td>
<td>142</td>
</tr>
<tr>
<td>initially_busy</td>
<td>Boolean string</td>
<td>true</td>
<td>no</td>
<td>false</td>
<td>142</td>
</tr>
<tr>
<td>peakwidth</td>
<td>double</td>
<td>0.2</td>
<td>-</td>
<td>0</td>
<td>142</td>
</tr>
<tr>
<td>period_busy</td>
<td>double</td>
<td>60</td>
<td>-</td>
<td>0</td>
<td>142</td>
</tr>
<tr>
<td>period_lazy</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>0</td>
<td>142</td>
</tr>
<tr>
<td>period_speed</td>
<td>double</td>
<td>0.8</td>
<td>-</td>
<td>0</td>
<td>142</td>
</tr>
<tr>
<td>reset_upon_running</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>true</td>
<td>142</td>
</tr>
<tr>
<td>summit_delta</td>
<td>double [0,100]</td>
<td>0.5</td>
<td>-</td>
<td>25</td>
<td>142</td>
</tr>
</tbody>
</table>

Table 68: Parameters for the BHV_PeriodicSpeed behavior.

Example Behavior File Configuration for BHV_PeriodicSpeed

Listing B.3 - An example BHV_PeriodicSpeed configuration.

```plaintext
0  Behavior = BHV_PeriodicSpeed
1  {
2     name = periodic_speed
3     priority = 500
4     period_length = 30 // Seconds
5     period_gap = 120 // Seconds
6     reset_on_idle = true // The default
7     initially_busy = false // The default
8     period_speed = 0.5 // Meters/sec
9     peakwidth = 0.3 // Meters/sec
10    basewidth = 0.5 // Meters/sec
11    summit_delta = 25 // The default
12  }
```

341
### Parameter Summary for BHV_PeriodicSurface

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sense</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>no starve</td>
<td>MOOSVAR, double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>acomms_mark_variable</td>
<td>MOOSVAR</td>
<td>RANGE_RECEIVED</td>
<td>yes</td>
<td>-</td>
<td>156</td>
</tr>
<tr>
<td>ascent_grade</td>
<td>string</td>
<td>quasi</td>
<td>no</td>
<td>linear</td>
<td>158</td>
</tr>
<tr>
<td>ascent_speed</td>
<td>double</td>
<td>1.0</td>
<td>-</td>
<td>*</td>
<td>158</td>
</tr>
<tr>
<td>mark_variable</td>
<td>MOOSVAR</td>
<td>GPS_RECEIVED</td>
<td>yes</td>
<td>GPS_UPDATE_RECEIVED</td>
<td>158</td>
</tr>
<tr>
<td>max_time_at_surface</td>
<td>MOOSVAR</td>
<td>60</td>
<td>yes</td>
<td>300</td>
<td>159</td>
</tr>
<tr>
<td>period</td>
<td>double</td>
<td>60</td>
<td>-</td>
<td>300</td>
<td>158</td>
</tr>
<tr>
<td>zero_speed_depth</td>
<td>double</td>
<td>2.5</td>
<td>-</td>
<td>0</td>
<td>159</td>
</tr>
</tbody>
</table>

Table 69: Parameters for the BHV_PeriodicSurface behavior.

### Example Behavior File Configuration for BHV_PeriodicSurface

**Listing B.4 - An example BHV_PeriodicSurface configuration.**

```plaintext
0  Behavior = BHV_PeriodicSurface
1  
2   name = bhv_periodic_surface
3   priority = 500
4   active_flag = SURFACING, IN_PROGRESS
5   inactive_flag = SURFACING, NO
6   
7       period = 3600  // seconds
8       ascent_speed = 1.0  // meters per second
9       zero_speed_depth = 2.5  // meters
10      max_time_at_surface = 120  // seconds
11      ascent_grade = linear
12      acomms_mark_variable = RANGE_RECEIVED
13      mark_variable = GPS_UPDATE_RECEIVED
14      status_variable = PERIODIC_PENDING_SURFACE
15  }
```

342
Parameter Summary for BHV_ConstantDepth

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>basewidth</td>
<td>double</td>
<td>15</td>
<td>-</td>
<td>100</td>
<td>162</td>
</tr>
<tr>
<td>depth</td>
<td>double</td>
<td>35</td>
<td>-</td>
<td>0</td>
<td>160</td>
</tr>
<tr>
<td>peakwidth</td>
<td>double</td>
<td>5</td>
<td>-</td>
<td>3</td>
<td>162</td>
</tr>
<tr>
<td>summitdelta</td>
<td>double</td>
<td>20</td>
<td>-</td>
<td>50</td>
<td>162</td>
</tr>
</tbody>
</table>

Table 70: Parameters for the BHV_ConstantDepth behavior.

Example Behavior File Configuration for BHV_ConstantDepth

Listing B.5 - An example BHV_ConstantDepth configuration.

```
0 Behavior = BHV_ConstantDepth
1 {
2    // General Behavior Parameters
3        name = constant_depth_survey
4        priority = 100
5        condition = AUTONOMY_MODE = SURVEY
6        duration = no-time-limit
7        updates = NEW_SURVEY_DEPTH
8        nostarve = NAV_DEPTH, 3.0
9
10    // BHV_ConstantDepth Behavior Parameters
11        depth = 50 // meters
12        peakwidth = 5
13        basewidth = 10
14        summitdelta = 45
15 }
```
Parameter Summary for BHV_ConstantHeading

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>basewidth</td>
<td>double</td>
<td>175</td>
<td>-</td>
<td>170</td>
<td>165</td>
</tr>
<tr>
<td>heading</td>
<td>double</td>
<td>35</td>
<td>-</td>
<td>0</td>
<td>165</td>
</tr>
<tr>
<td>peakwidth</td>
<td>double</td>
<td>5</td>
<td>-</td>
<td>10</td>
<td>165</td>
</tr>
<tr>
<td>summitdelta</td>
<td>double</td>
<td>35</td>
<td>-</td>
<td>25</td>
<td>165</td>
</tr>
</tbody>
</table>

Table 71: Parameters for the BHV_ConstantHeading behavior.

Example Behavior File Configuration for BHV_ConstantHeading

Listing B.6 - An example BHV_ConstantHeading configuration.

```plaintext
0 Behavior = BHV_ConstantHeading
1 {
2   name = bhv_constant_heading
3   priority = 100
4   duration = 60
5   condition = AUTONOMY_MODE = PID_TEST
6   updates = NEW_TEST_HEADING
7   nostarve = NAV_HEADING, 3.0
8     heading = 45 // degrees
9   peakwidth = 30
10  basewidth = 150
11  summitdelta = 25
12 }
```

344
Parameter Summary for BHV_ConstantSpeed

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td></td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>basewidth</td>
<td>double</td>
<td>0.6</td>
<td>-</td>
<td>2.0</td>
<td>168</td>
</tr>
<tr>
<td>speed</td>
<td>double</td>
<td>1.2</td>
<td>-</td>
<td>0.0</td>
<td>168</td>
</tr>
<tr>
<td>peakwidth</td>
<td>double</td>
<td>0.1</td>
<td>-</td>
<td>0.0</td>
<td>168</td>
</tr>
<tr>
<td>summitdelta</td>
<td>double</td>
<td>35</td>
<td>-</td>
<td>0</td>
<td>168</td>
</tr>
</tbody>
</table>

Table 72: Parameters for the BHV_ConstantSpeed behavior.

Example Behavior File Configuration for BHV_ConstantSpeed

Listing B.7 - An example BHV_ConstantSpeed configuration.

```
0 Behavior = BHV_ConstantSpeed
1 {
2   name = const_speed_bravo
3   priority = 100
4   duration = 60
5   active_flag = BRAVO_SPEED_TEST = in-progress
6   nostarve = NAV_SPEED, 2.0
7
8   speed = 1.8 // meters per second
9   peakwidth = 0.3
10   basewidth = 1.0
11   summitdelta = 22
12 }
```
Parameter Summary for BHV_GoToDepth

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostrarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>depth, depths</td>
<td>string</td>
<td>50,10:40,60</td>
<td>yes</td>
<td>-</td>
<td>171</td>
</tr>
<tr>
<td>repeat</td>
<td>int</td>
<td>5</td>
<td>-</td>
<td>0</td>
<td>171</td>
</tr>
<tr>
<td>capture_delta</td>
<td>double</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>171</td>
</tr>
<tr>
<td>capture_flag</td>
<td>MOOSVAR</td>
<td>DEPTH_HIT</td>
<td>yes</td>
<td>-</td>
<td>171</td>
</tr>
</tbody>
</table>

Table 73: Parameters for the BHV_GoToDepth behavior.

Example Behavior File Configuration for BHV_GoToDepth

Listing B.8 - An example BHV_GoToDepth configuration.

```
0  Behavior = BHV_GoToDepth
1  {
2   name = goto_depth_set_alpha
3   priority = 100
4   condition = DEPLOY == true
5   endflag = GOTO_DEPTH_ALPHA = DONE
6   depths = 15,30: 30,30: 45,60: 15,30
7   capture_delta = 1 // meters
8   capture_flag = DEPTH_LEVELS_ACHIEVED
9   repeat = 4
10  }
```
Parameter Summary for BHV_MemoryTurnLimit

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean,string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>memory_time</td>
<td>double</td>
<td>60</td>
<td>-</td>
<td>-1</td>
<td>173</td>
</tr>
<tr>
<td>turn_range</td>
<td>double</td>
<td>45</td>
<td>-</td>
<td>-1</td>
<td>173</td>
</tr>
</tbody>
</table>

Table 74: Parameters for the BHV_MemoryTurnLimit behavior.

Example Behavior File Configuration for BHV_MemoryTurnLimit

Listing B.9 - An example BHV_MemoryTurnLimit configuration.

```plaintext
0 Behavior = BHV_MemoryTurnLimit
1 {
2   name = memturnlimit
3   priority = 1000
4   memory_time = 60 // seconds
5   turn_range = 35 // degrees
6 }
```

347
Parameter Summary for BHV_StationKeep

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>yes</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>yes</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,MOOSVAR</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>station_pt, point</td>
<td>string</td>
<td>50,75</td>
<td>yes</td>
<td>0,0</td>
<td>178</td>
</tr>
<tr>
<td>center_activate</td>
<td>Boolean string</td>
<td>TRUE</td>
<td>no</td>
<td>false</td>
<td>178</td>
</tr>
<tr>
<td>inner_radius</td>
<td>double</td>
<td>10</td>
<td>-</td>
<td>4</td>
<td>179</td>
</tr>
<tr>
<td>outer_radius</td>
<td>double</td>
<td>25</td>
<td>-</td>
<td>15</td>
<td>179</td>
</tr>
<tr>
<td>outer_speed</td>
<td>double</td>
<td>1.8</td>
<td>-</td>
<td>1.2</td>
<td>179</td>
</tr>
<tr>
<td>transit_speed</td>
<td>double</td>
<td>1.9</td>
<td>-</td>
<td>2.5</td>
<td>176</td>
</tr>
<tr>
<td>passive_station_radius</td>
<td>double</td>
<td>200</td>
<td>-</td>
<td>0</td>
<td>176</td>
</tr>
<tr>
<td>passive_station_variable</td>
<td>MOOSVAR</td>
<td>PSK_MODE_CHARLIE</td>
<td>-</td>
<td>PSKEEP_MODE</td>
<td>176</td>
</tr>
</tbody>
</table>

Table 75: Parameters for the BHV_StationKeep behavior.

Example Behavior File Configuration for BHV_StationKeep

Listing B.10 - An example BHV_StationKeep configuration.

```plaintext
0 Behavior = BHV_StationKeep
1 {
2   name = bhv_station_keep
3   priority = 100
4   condition = (ON_STATION=true) and (RETURN=false)
5   updates = STATION_UPDATES
6     station_pt = 200,-150
7     center_activate = true
8     inner_radius = 10
9     outer_radius = 40
10    outer_speed = 0.8
11    transit_speed = 1.8
12    passive_station_radius = 400 // meters
13    passive_station_variable = PSKEEP_MODE // the default
14 }```

348
Parameter Summary for BHV_Timer

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
</tbody>
</table>

No additional parameters for this behavior

Table 76: Parameters for the BHV_Timer behavior.

Example Behavior File Configuration for BHV_Timer

Listing B.11 - An example BHV_Timer configuration.

```plaintext
0 Behavior = BHV_Timer
1 {
2   name = bhv_timer_a
3   duration = 60     // seconds
4   condition = loiter = alpha
5   end_flag = loiter = beta
6 }
```

349
Parameter Summary for BHV_AvoidCollision

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>contact</td>
<td>string</td>
<td>Alliance</td>
<td>yes</td>
<td>-</td>
<td>197</td>
</tr>
<tr>
<td>on_no_contact_ok</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>true</td>
<td>198</td>
</tr>
<tr>
<td>extrapolate</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>false</td>
<td>197</td>
</tr>
<tr>
<td>decay</td>
<td>double,double</td>
<td>10, 30</td>
<td>yes</td>
<td>-</td>
<td>197</td>
</tr>
<tr>
<td>bearing_lines</td>
<td>string</td>
<td>quadratic</td>
<td>no</td>
<td>linear</td>
<td>201</td>
</tr>
<tr>
<td>pwt_grade</td>
<td>string</td>
<td>quadratic</td>
<td>no</td>
<td>linear</td>
<td>206</td>
</tr>
<tr>
<td>completed_dist</td>
<td>double</td>
<td>200</td>
<td>-</td>
<td>500</td>
<td>201</td>
</tr>
<tr>
<td>pwt_inner_dist</td>
<td>double</td>
<td>50</td>
<td>-</td>
<td>50</td>
<td>206</td>
</tr>
<tr>
<td>pwt_outer_dist</td>
<td>double</td>
<td>200</td>
<td>-</td>
<td>200</td>
<td>206</td>
</tr>
<tr>
<td>max_util_cpa_dist</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>75</td>
<td>207</td>
</tr>
<tr>
<td>min_util_cpa_dist</td>
<td>double</td>
<td>10</td>
<td>-</td>
<td>10</td>
<td>207</td>
</tr>
</tbody>
</table>

Table 77: Parameters for the BHV_AvoidCollision behavior.

Example Behavior File Configuration for BHV_AvoidCollision

Listing B.12 - An example BHV_AvoidCollision configuration.

```plaintext
0 Behavior = BHV_AvoidCollision
1 {
2   name = avoid_collision_alpha
3   pwt = 100
4   condition = AVOIDANCE_MODE != INACTIVE
5     contact = alpha
6     on_no_contact_ok = true
7     extrapolate = true
8     decay = 30,60
9
10   pwt_outer_dist = 150
11   pwt_inner_dist = 75
12   min_util_cpa_dist = 15
13   max_util_cpa_dist = 80
14   pwt_grade = linear
```

350
16  bearing_line_config = white:0, green:0.65, yellow:0.8, red:1.016
14  }
Parameter Summary for BHV_CutRange

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>-</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>-</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>contact</td>
<td>string</td>
<td>Alliance</td>
<td>yes</td>
<td>-</td>
<td>197</td>
</tr>
<tr>
<td>on_no_contact_ok</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>true</td>
<td>198</td>
</tr>
<tr>
<td>extrapolate</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>false</td>
<td>197</td>
</tr>
<tr>
<td>decay</td>
<td>double,double</td>
<td>10, 30</td>
<td>-</td>
<td>0,0</td>
<td>197</td>
</tr>
<tr>
<td>dist_priority_interval</td>
<td>double,double</td>
<td>40,100</td>
<td>-</td>
<td>0,0</td>
<td>211</td>
</tr>
<tr>
<td>time_on_leg</td>
<td>double</td>
<td>60</td>
<td>-</td>
<td>15</td>
<td>211</td>
</tr>
<tr>
<td>give_up_range</td>
<td>double</td>
<td>500</td>
<td>-</td>
<td>0</td>
<td>211</td>
</tr>
<tr>
<td>patience</td>
<td>double</td>
<td>50</td>
<td>-</td>
<td>0</td>
<td>211</td>
</tr>
</tbody>
</table>

Table 78: Parameters for the BHV_CutRange behavior.

Example Behavior File Configuration for BHV_CutRange

Listing B.13 - An example BHV_CutRange configuration.

```
0 Behavior = BHV_CutRange
1 {
2   name = bhv_cutrange
3   pwt = 100
4   contact = zulu
5
6   dist_priority_interval = 25,100
7   time_on_leg = 60
8   give_up_range = 400
9   patience = 75
10 }
```
Parameter Summary for BHV_Shadow

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td>yes</td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority, pwt</td>
<td>double</td>
<td>100</td>
<td>yes</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idleflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR,double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR,MOOSVAR</td>
<td>FOO,BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>contact</td>
<td>string</td>
<td>Alliance</td>
<td>yes</td>
<td>-</td>
<td>197</td>
</tr>
<tr>
<td>on_no_contact_ok</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>true</td>
<td>198</td>
</tr>
<tr>
<td>extrapolate</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>false</td>
<td>197</td>
</tr>
<tr>
<td>decay</td>
<td>double,double</td>
<td>10, 30</td>
<td>yes</td>
<td>-</td>
<td>197</td>
</tr>
<tr>
<td>max_range</td>
<td>double</td>
<td>100</td>
<td>yes</td>
<td>-</td>
<td>217</td>
</tr>
<tr>
<td>heading_peakwidth</td>
<td>double</td>
<td>10</td>
<td>yes</td>
<td>-</td>
<td>217</td>
</tr>
<tr>
<td>heading_basewidth</td>
<td>double</td>
<td>170</td>
<td>yes</td>
<td>-</td>
<td>217</td>
</tr>
<tr>
<td>speed_peakwidth</td>
<td>double</td>
<td>0.3</td>
<td>yes</td>
<td>-</td>
<td>217</td>
</tr>
<tr>
<td>speed_basewidth</td>
<td>double</td>
<td>0.5</td>
<td>yes</td>
<td>-</td>
<td>217</td>
</tr>
</tbody>
</table>

Table 79: Parameters for the BHV_Shadow behavior.

Example Behavior File Configuration for BHV_Shadow

Listing B.14 - An example BHV_Shadow configuration.

0 Behavior = BHV_Shadow
1 {
2   name = bhv_shadow
3   pwt = 100
4   contact = delta
5
6   max_range = 200
7   heading_peakwidth = 10
8   heading_basewidth = 170
9   speed_peakwidth = 10
10  speed_basewidth = 170
11 }

353
## Parameter Summary for BHV_Trail

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument Type</th>
<th>Example</th>
<th>Case-Sensitive</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>loiter-west-zone</td>
<td>yes</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>duration</td>
<td>double</td>
<td>600</td>
<td></td>
<td>-1</td>
<td>102</td>
</tr>
<tr>
<td>duration_status</td>
<td>MOOSVAR</td>
<td>loiter_remaining</td>
<td>yes</td>
<td>-</td>
<td>102</td>
</tr>
<tr>
<td>priority; pwt</td>
<td>double</td>
<td>100</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>runflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = maybe</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>endflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = done</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>activeflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = yes</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>inactiveflag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = off</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>idletag</td>
<td>MOOSVAR=value</td>
<td>LOITERING = no</td>
<td>yes</td>
<td>-</td>
<td>89</td>
</tr>
<tr>
<td>nostarve</td>
<td>MOOSVAR, double</td>
<td>INFO,60</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>perpetual</td>
<td>Boolean string</td>
<td>false</td>
<td>no</td>
<td>false</td>
<td>103</td>
</tr>
<tr>
<td>post_mapping</td>
<td>MOOSVAR, MOOSVAR</td>
<td>FOO, BAR</td>
<td>yes</td>
<td>-</td>
<td>103</td>
</tr>
<tr>
<td>updates</td>
<td>MOOSVAR</td>
<td>LOITER_INFO</td>
<td>yes</td>
<td>-</td>
<td>101</td>
</tr>
<tr>
<td>condition</td>
<td>Logic Expression</td>
<td>QUALITY &lt;= 7</td>
<td>yes</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>contact</td>
<td>string</td>
<td>Alliance</td>
<td>yes</td>
<td>-</td>
<td>197</td>
</tr>
<tr>
<td>on_no_contact_ok</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>true</td>
<td>198</td>
</tr>
<tr>
<td>extrapolate</td>
<td>boolean</td>
<td>true</td>
<td>no</td>
<td>false</td>
<td>197</td>
</tr>
<tr>
<td>decay</td>
<td>double, double</td>
<td>10, 30</td>
<td>-</td>
<td>0,0</td>
<td>197</td>
</tr>
<tr>
<td>trail_range</td>
<td>double</td>
<td>20</td>
<td>-</td>
<td>50</td>
<td>222</td>
</tr>
<tr>
<td>trail_angle</td>
<td>double</td>
<td>270</td>
<td>-</td>
<td>180</td>
<td>222</td>
</tr>
<tr>
<td>trail_angle_type</td>
<td>string</td>
<td>absolute</td>
<td>no</td>
<td>relative</td>
<td>222</td>
</tr>
<tr>
<td>radius</td>
<td>double</td>
<td>8</td>
<td>-</td>
<td>5</td>
<td>222</td>
</tr>
<tr>
<td>nm_radius</td>
<td>double</td>
<td>20</td>
<td>-</td>
<td>20</td>
<td>222</td>
</tr>
<tr>
<td>max_range</td>
<td>double</td>
<td>50</td>
<td>-</td>
<td>0</td>
<td>222</td>
</tr>
</tbody>
</table>

Table 80: Parameters for the BHV_Trail behavior.

## Example Behavior File Configuration for BHV_Trail

Listing B.15 - An example BHV_Trail configuration.

```plaintext
0 Behavior = BHV_Trail
1 {
2    name = bhv_trail
3    priority = 100
4    contact = delta
5    extrapolate = true
6    on_no_contact_ok = true
7    decay = 20,60 // seconds
8
9    trail_range = 50 // meters
10   trail_angle = 185 // degrees
11   trail_angle_type = relative
12   radius = 10 // meters
13   nm_radius = 30 // meters
14   max_range = 300 // meters
15 }
```

354
C  Colors

Below are the colors used by IvP utilities that use colors. Colors are case insensitive. A color may be specified by the string as shown, or with the '_' character as a separator. Or the color may be specified with its hexadecimal or floating point form. For example the following are equivalent:

"darkblue", "DarkBlue", "dark_blue", "hex:00,00,8b", and "0,0,0.545".

In the latter two styles, the '%', '$', or '#' characters may also be used as a delimiter instead of the comma if it helps when embedding the color specification in a larger string that uses its own delimiters. Mixed delimiters are not supported however.

In most cases, the colors invisible, empty, off are aliases indicating that the relevant object, line or vertex etc is not to be rendered at all.

<table>
<thead>
<tr>
<th>Color Name</th>
<th>RGB Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>antiquewhite</td>
<td>(fa,eb,d7)</td>
</tr>
<tr>
<td>aquamarine</td>
<td>(7f,ff,d4)</td>
</tr>
<tr>
<td>beige</td>
<td>(f5,f5,dc)</td>
</tr>
<tr>
<td>black</td>
<td>(00,00,00)</td>
</tr>
<tr>
<td>blue</td>
<td>(00,00,ff)</td>
</tr>
<tr>
<td>brown</td>
<td>(a5,2a,2a)</td>
</tr>
<tr>
<td>cadetblue</td>
<td>(5f,9e,a0)</td>
</tr>
<tr>
<td>chocolate</td>
<td>(d2,69,1e)</td>
</tr>
<tr>
<td>cornsilk</td>
<td>(ff,88,dc)</td>
</tr>
<tr>
<td>crimson</td>
<td>(de,14,3c)</td>
</tr>
<tr>
<td>darkblue</td>
<td>(00,00,8b)</td>
</tr>
<tr>
<td>darkgoldenrod</td>
<td>(b8,86,0b)</td>
</tr>
<tr>
<td>darkgreen</td>
<td>(00,64,00)</td>
</tr>
<tr>
<td>darkmagenta</td>
<td>(8b,00,8b)</td>
</tr>
<tr>
<td>darkorange</td>
<td>(ff,8c,00)</td>
</tr>
<tr>
<td>darkred</td>
<td>(8b,00,00)</td>
</tr>
<tr>
<td>darkseagreen</td>
<td>(8f,bc,8f)</td>
</tr>
<tr>
<td>darkslategray</td>
<td>(2f,4f,4f)</td>
</tr>
<tr>
<td>darkviolet</td>
<td>(94,00,d3)</td>
</tr>
<tr>
<td>deepskyblue</td>
<td>(00,bf,ff)</td>
</tr>
<tr>
<td>dodgerblue</td>
<td>(1e,90,ff)</td>
</tr>
<tr>
<td>floralwhite</td>
<td>(ff,fa,f0)</td>
</tr>
<tr>
<td>fuchsia</td>
<td>(ff,00,ff)</td>
</tr>
<tr>
<td>ghostwhite</td>
<td>(f8,f8,ff)</td>
</tr>
<tr>
<td>goldenrod</td>
<td>(da,a5,20)</td>
</tr>
<tr>
<td>green</td>
<td>(00,80,00)</td>
</tr>
<tr>
<td>honeydew</td>
<td>(f0,ff,f0)</td>
</tr>
<tr>
<td>indianred</td>
<td>(cd,5c,5c)</td>
</tr>
<tr>
<td>aqua</td>
<td>(00,ff,ff)</td>
</tr>
<tr>
<td>azure</td>
<td>(f0,ff,ff)</td>
</tr>
<tr>
<td>bisque</td>
<td>(ff,e4,c4)</td>
</tr>
<tr>
<td>blanchedalmond</td>
<td>(ff,eb,cd)</td>
</tr>
<tr>
<td>blueviolet</td>
<td>(8a,2b,e2)</td>
</tr>
<tr>
<td>burlywood</td>
<td>(de,b8,87)</td>
</tr>
<tr>
<td>chartreuse</td>
<td>(7f,ff,00)</td>
</tr>
<tr>
<td>coral</td>
<td>(ff,7f,50)</td>
</tr>
<tr>
<td>cornflowerblue</td>
<td>(64,95,ed)</td>
</tr>
<tr>
<td>cyan</td>
<td>(00,ff,ff)</td>
</tr>
<tr>
<td>darkcyan</td>
<td>(00,8b,8b)</td>
</tr>
<tr>
<td>darkgray</td>
<td>(a9,a9,a9)</td>
</tr>
<tr>
<td>darkkhaki</td>
<td>(bd,b7,6b)</td>
</tr>
<tr>
<td>darkolivegreen</td>
<td>(55,6b,2f)</td>
</tr>
<tr>
<td>darkorchid</td>
<td>(99,32,cc)</td>
</tr>
<tr>
<td>darksalmon</td>
<td>(e9,96,7a)</td>
</tr>
<tr>
<td>darkslateblue</td>
<td>(48,3d,8b)</td>
</tr>
<tr>
<td>darkturquoise</td>
<td>(00,ce,d1)</td>
</tr>
<tr>
<td>deeppink</td>
<td>(ff,14,93)</td>
</tr>
<tr>
<td>dingray</td>
<td>(69,69,69)</td>
</tr>
<tr>
<td>firebrick</td>
<td>(b2,22,22)</td>
</tr>
<tr>
<td>forestgreen</td>
<td>(22,8b,22)</td>
</tr>
<tr>
<td>gainsboro</td>
<td>(dc,dc,dc)</td>
</tr>
<tr>
<td>gold</td>
<td>(ff,d7,00)</td>
</tr>
<tr>
<td>gray</td>
<td>(80,80,80)</td>
</tr>
<tr>
<td>greenyellow</td>
<td>(ad,ff,2f)</td>
</tr>
<tr>
<td>hotpink</td>
<td>(ff,69,b4)</td>
</tr>
<tr>
<td>indigo</td>
<td>(4b,00,82)</td>
</tr>
</tbody>
</table>
ivory (ff,ff,f0)
lavender (e6,e6,fa)
lawngreen (7c,fc,00)
lightblue (ad,d8,e6)
lightcyan (e0,ff,ff)
lightgray (d3,d3,d3)
lightpink (ff,b6,c1)
lightseagreen (20,b2,aa)
lightslategray (77,88,99)
lightyellow (ff,ff,e0)
limegreen (32,cd,32)
magenta (ff,00,ff)
mediumblue (00,00,cd)
mediumseagreen (3c,b3,71)
mediumspringgreen (00,fa,9a)
mediumvioletred (c7,15,85)
mintcream (f5,ff,fa)
mocassin (ff,e4,b5)
navy (00,00,80)
olive (80,80,00)
orange (ff,a5,00)
orchid (da,70,d6)
paleturquoise (af,ee,ee)
papayawhip (ff,ef,d5)
pelegoldenrod (ee,e8,aa)
pink (ff,c0,cb)
powderblue (b0,e0,e6)
red (ff,00,00)
royalblue (41,69,e1)
salmon (fa,80,72)
seagreen (2e,8b,57)
sienna (a0,52,2d)
skyblue (87,ce,eb)
slategray (70,80,90)
springgreen (00,ff,7f)
tan (d2,b4,8c)
thistle (d8,bf,d8)
turquoise (40,e0,d0)
wheat (f5,de,b3)
whitesmoke (f5,f5,f5)
khaki (f0,e6,8c)
lavenderblush (ff,f0,f5)
lemonchiffon (ff,fa,cd)
lightcoral (f0,80,80)
lightgoldenrod (fa,fa,d2)
lightgreen (90,ee,90)
lightsalmon (ff,a0,7a)
lightskyblue (87,ee,fa)
lightsteelblue (b0,c4,de)
lime (00,ff,00)
linen (fa,f0,06)
maroon (80,00,00)
mediumorchid (ba,55,d3)
mediumslateblue (7b,68,ee)
mediumturquoise (48,d1,cc)
midnightblue (19,19,70)
mistyrose (ff,e4,e1)
navajowhite (ff,de,ad)
oldlace (fd,f5,e6)
olivedrab (6b,8e,23)
orangered (ff,45,00)
palegreen (98,fb,98)
palevioletred (db,70,93)
peachpuff (ff,da,b9)
peru (cd,85,3f)
plum (dd,a0,dd)
purple (80,00,80)
rosybrown (bc,8f,8f)
saddlebrown (8b,45,13)
sandybrown (f4,a4,60)
seashell (ff,f5,ee)
silver (c0,c0,c0)
slateblue (6a,5a,cd)
snow (ff,fa,fa)
steelblue (46,82,b4)
teal (00,80,80)
tomatao (ff,63,47)
violet (ee,82,ee)
white (ff,ff,ff)
yellow (ff,ff,00)
yellowgreen (9a,cd,32)
References


Index

Action Selection, 78
addComponent(), 298
addInfoVars(), 107
All-Stop, 64
Alpha Example Mission, 54
Antler, 58
AOF Class, 308
   Class Definition, 308
   Example Implementation, 308, 309
   Gaussian Functions, 312
   General Properties, 308
atof(), 104
AutoPeak Refinement, 291, 316, 330
   Advantages and Disadvantages, 330
   Brief Summary, 291
   Detailed Usage, 330
AvoidCollision Behavior, 201
AvoidObstacle Behavior, 146
Backseat Driver, 34
Behavior Files, 81
   Loading, 66
   Syntax checking, 69
   Variable Initialization, 81
Behavior Subscriptions, 61
Behavior-Posts, 75
Berta Example Mission, 276
bool isComplete(), 106
bool isRunnable(), 106
bool setBaseWidth(double basewidth), 302
bool setBaseWidth(double val, int index=0), 295
bool setMinMaxUtil(double min, double max), 302
bool setMinMaxUtil(double min, double max, int index=0), 295
bool setStateOK(), 296, 302, 315
 Bravo Example Mission, 84
buildFunctionWithReflector(), 339, 343
buildFunctionWithZAIC(), 339, 340
Charlie Example Mission, 247
checkUpdates(), 102, 107
Conditions, 347
ConstantDepth Behavior, 160, 256
ConstantHeading Behavior, 163
ConstantSpeed Behavior, 166
Contact Management, 279
create(), 314–316, 321–323, 330
CutRange Behavior, 211
Delta Example Mission, 255
Directed Refinement, 291, 316, 322
   Brief Summary, 291
   Detailed Usage, 322
double getBufferCurrTime(), 107
double getBufferDoubleVal(string varname, bool&
result), 107
double getBufferStringVal(string varname, bool&
result), 107
DRIVE, 62
Duplication Filter, 74, 80, 339
Dynamic Behavior Spawning, 111, 265, 277
Echo Example Mission, 264, 270
evalPoint(), 308–310
Example Missions
   Mission M2: The Berta Mission, 276
   Mission S11: The Kilo Mission, 270
Mission S1: The Alpha Mission, 54
Mission S2: The Bravo Mission, 84
Mission S3: The Charlie Mission, 247
Mission S4: The Delta Mission, 255
Mission S5: The Echo Mission, 264

extract(), 309
extractIvPFunction(), 314

Geometry Utilities
   Ellipses, 247
   SegLists, 258
getBuffer*(), 376
getBufferDoubleVal(), 110, 338
getBufferStringVal(), 110
getWarnings(), 315, 340
GoToDepth Behavior, 169

Helm State, 62
   START_IN_DRIVE, 64
   start_in_drive, 67
   On Helm Start-Up, 64
   Relation to All-Stop, 66
   Transitions, 62
Hierarchical Mode Declarations, 83, 88, 247
   Example Mission, 84
   Realizable Modes, 86
   Run Time Monitoring, 87
   Syntax, 86
initialize(), 308
int addComponent(), 295
int create(int pieces=-1), 314
isNumber(), 104
isRunnable(), 290
Iterate(), 40–42, 44–47, 49, 79–82, 87, 89, 110
IvP Behavior Functions
   Helm-Invoked Functions, 104
   Helm-Invoked Immutable Functions, 106
   Helm-Invoked Overloadable Functions, 106
   Implementor-Invoked Functions, 107
   The checkUpdates() Function, 106
   The getBuffer*() Functions, 107, 109
   The getBufferCurrTime() Function, 107
   The isComplete() Function, 106
   The isRunnable() Function, 106
   The onIdleState() Function, 96, 106, 110
   The onIdleToRunState() Function, 96
   The onRunState() Function, 96, 106, 110
   The post*Message() Functions, 108
   The postFlags() Function, 106
   The setComplete() Function, 107
   The spawnflag, 90
   The updates, 101, 267
IvP Behaviors, 80, 87
   Conditions, see Run Conditions
   Duration, 89
   Dynamic Configuration, 101
   Flags, 89
   Independence, 77
   Messages, 89
   Priority Weights, 94
   Run Conditions, 88
   Run States, 89, 91, 339
   Sequences, 78
   State, 78
   Subscriptions, 61
   The Information Buffer, 108
IvP Behaviors Implemented
   BHV_AvoidCollision, 201, 277
By Choosing the Piece Count, 319
By Choosing the Piece Shape, 321
Choosing the Piece Shape, 319
uSimMarine, 59
uTimerScript
    Example Usage, 112
Value Functions, 287
vector<double> getBufferDoubleVector(string var, bool& result), 107
vector<string> getBufferStringVector(string var, bool& result), 108
void addInfoVars(string var_names), 107
void checkUpdates(), 106
void onIdleState(), 106
void postBoolMessage(string varname, bool value, string key), 108
void postEMessage(string error_msg), 108
void postFlags(string flag_type), 106
void postIntMessage(string varname, double value, string key), 108
void postMessage(string varname, double value, string key), 108
void postMessage(string varname, string value, string key), 108
void postRepeatableMessage(string varname, double value), 108
void postRepeatableMessage(string varname, string value), 108
void postWMessage(string warning_msg), 108
void setComplete(), 107
void setSummitInsist(bool val), 295
void setValueWrap(bool val), 295
Waypoint Behavior, 116, 258
Wind - Simulated Wind Gusts, 247
ZAIC Tools, 293
    General Properties, 293
    Maximum Utility Plateau, 304
ZAIC_HEQ Tool, 300, 303
ZAIC_LEQ Tool, 300
    Example Usage, 302
    Interface, 300
ZAIC_LEQ(IvPDomain domain, string varname), 301
ZAIC_PEAK Tool, 293
    basewidth parameter, 295
    maxutil parameter, 295
    minutil parameter, 295
    peakwidth parameter, 295
    summitdelta parameter, 295
    summitinsist parameter, 295–297
    summit parameter, 295
    valuewrap parameter, 295, 296
    Example Usage, 297, 340
    Extracting an IvPFunction, 296
    Extracting an IvPFunction, 341
    Interface, 293, 294
    Multi-Modal Functions, 298
    The basewidth parameter, 340
    The peakwidth parameter, 340
    The summitdelta parameter, 340
    The summit parameter, 340
ZAIC_Vector Tool, 304
    Example Usage, 305
## Index of MOOS Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>=PARK*, 73</td>
<td></td>
</tr>
<tr>
<td>AVD_OB_spawn, 149</td>
<td></td>
</tr>
<tr>
<td>BCM_ALERT_REQUEST, 204, 206</td>
<td></td>
</tr>
<tr>
<td>BEARING_POINT, 266, 267</td>
<td></td>
</tr>
<tr>
<td>BHV_ERROR, 64, 65, 71, 108, 198, 290</td>
<td></td>
</tr>
<tr>
<td>BHV_WARNING, 71, 102, 108, 121, 179, 198, 303, 306</td>
<td></td>
</tr>
<tr>
<td>CLOSING_SPD_AVD, 204</td>
<td></td>
</tr>
<tr>
<td>CONTACT_INFO, 277–279</td>
<td></td>
</tr>
<tr>
<td>CONTACT_RESOLVED, 204</td>
<td></td>
</tr>
<tr>
<td>CONVOY_RECAP, 237, 240</td>
<td></td>
</tr>
<tr>
<td>CONVOY_SPD_POLICY, 236</td>
<td></td>
</tr>
<tr>
<td>CONVOY_STAT_RECAP, 237</td>
<td></td>
</tr>
<tr>
<td>CONVOY_UPDATES, 232</td>
<td></td>
</tr>
<tr>
<td>CREATE_CPU, 71, 95</td>
<td></td>
</tr>
<tr>
<td>CYCLE_INDEX, 118, 119, 123, 124</td>
<td></td>
</tr>
<tr>
<td>DB_CLIENTS, 50</td>
<td></td>
</tr>
<tr>
<td>DEPLOY, 57, 60</td>
<td></td>
</tr>
<tr>
<td>DESIRED_*, 71</td>
<td></td>
</tr>
<tr>
<td>DESIRED_DEPTH=0, 64</td>
<td></td>
</tr>
<tr>
<td>DESIRED_DEPTH, 169</td>
<td></td>
</tr>
<tr>
<td>DESIRED_HEADING, 59, 73, 383</td>
<td></td>
</tr>
<tr>
<td>DESIRED_RUDDER, 59, 60</td>
<td></td>
</tr>
<tr>
<td>DESIRED_SPEED=0, 64</td>
<td></td>
</tr>
<tr>
<td>DESIRED_SPEED, 59, 74</td>
<td></td>
</tr>
<tr>
<td>DESIRED_THRUST, 59, 60</td>
<td></td>
</tr>
<tr>
<td>FOO_DESIRED_SPEED, 65</td>
<td></td>
</tr>
<tr>
<td>GPS_UPDATE_RECEIVED, 158, 258</td>
<td></td>
</tr>
<tr>
<td>GPS_X, 40</td>
<td></td>
</tr>
<tr>
<td>GPS_Y, 40</td>
<td></td>
</tr>
<tr>
<td>HELM_IPF_COUNT, 71</td>
<td></td>
</tr>
<tr>
<td>HELM_MAP_CLEAR, 71, 75, 76</td>
<td></td>
</tr>
<tr>
<td>HELM_VERBOSE, 68, 71</td>
<td></td>
</tr>
<tr>
<td>IVPHELM_ALLSTOP, 64, 65, 260</td>
<td></td>
</tr>
<tr>
<td>IVPHELM_DOMAIN, 71</td>
<td></td>
</tr>
<tr>
<td>IVPHELM_LIFE_EVENT, 112, 113</td>
<td></td>
</tr>
<tr>
<td>IVPHELM_MODESET, 71</td>
<td></td>
</tr>
<tr>
<td>IVPHELM_STATEVARS, 70</td>
<td></td>
</tr>
<tr>
<td>IVPHELM_STATE, 62, 63, 71, 73, 74, 260, 272, 274, 275</td>
<td></td>
</tr>
<tr>
<td>IVPHELM_SUMMARY, 70, 91, 248</td>
<td></td>
</tr>
<tr>
<td>LOITER_ACQUIRE, 135, 138</td>
<td></td>
</tr>
<tr>
<td>LOITER_DIST2POLY, 138</td>
<td></td>
</tr>
<tr>
<td>LOITER_DIST_TO_POLY, 135</td>
<td></td>
</tr>
<tr>
<td>LOITER_ETA_TO_POLY, 135</td>
<td></td>
</tr>
<tr>
<td>LOITER_INDEX_FLAG, 138</td>
<td></td>
</tr>
<tr>
<td>LOITER_INDEX, 135, 138</td>
<td></td>
</tr>
<tr>
<td>LOITER_MODE, 135, 141</td>
<td></td>
</tr>
<tr>
<td>LOITER_REPORT, 135, 138, 249</td>
<td></td>
</tr>
<tr>
<td>LOOP_CPU, 71, 95</td>
<td></td>
</tr>
<tr>
<td>MAX_RANGE, 225</td>
<td></td>
</tr>
<tr>
<td>MEM_TURN_AVG, 175</td>
<td></td>
</tr>
<tr>
<td>MODE, 80, 85</td>
<td></td>
</tr>
<tr>
<td>MOOS_MANUAL_OVERRIDE=false, 63</td>
<td></td>
</tr>
<tr>
<td>MOOS_MANUAL_OVERRIDE, 62, 63, 65–67, 71</td>
<td></td>
</tr>
<tr>
<td>MOOS_MANUAL_OVERRIDE=true, 73</td>
<td></td>
</tr>
<tr>
<td>MOOS_MANUAL_OVERRIDE, 63, 67, 248</td>
<td></td>
</tr>
<tr>
<td>MVIEWER_LCLICK, 383</td>
<td></td>
</tr>
<tr>
<td>MVIEWER_RCLICK, 383</td>
<td></td>
</tr>
<tr>
<td>NAV_DEPTH, 60, 171</td>
<td></td>
</tr>
<tr>
<td>NAV_HEADING, 60</td>
<td></td>
</tr>
<tr>
<td>NAV_SPEED, 60</td>
<td></td>
</tr>
<tr>
<td>NAV_X, 60, 130</td>
<td></td>
</tr>
<tr>
<td>NAV_Y, 60, 130</td>
<td></td>
</tr>
<tr>
<td>NODE_REPORT_LOCAL, 60</td>
<td></td>
</tr>
<tr>
<td>NODE_REPORT, 60, 189, 191, 197, 279</td>
<td></td>
</tr>
<tr>
<td>NOTED_RESOLVED, 150</td>
<td></td>
</tr>
<tr>
<td>OPREG_ABSOLUTE_PERIM_DIST, 129, 131</td>
<td></td>
</tr>
<tr>
<td>OPREG_ABSOLUTE_PERIM_ETA, 129, 131</td>
<td></td>
</tr>
<tr>
<td>OPREG_TIME_REMAINING, 129</td>
<td></td>
</tr>
<tr>
<td>OPREG_TRAJECTORY_PERIM_DIST, 129, 131</td>
<td></td>
</tr>
<tr>
<td>OPREG_TRAJECTORY_PERIM_ETA, 129, 131</td>
<td></td>
</tr>
<tr>
<td>OPR_RESET = poly, 132</td>
<td></td>
</tr>
<tr>
<td>OPR_RESET=altitude, 132</td>
<td></td>
</tr>
<tr>
<td>OPR_RESET=depth, 132</td>
<td></td>
</tr>
<tr>
<td>OPR_RESET=time, 132</td>
<td></td>
</tr>
<tr>
<td>PENDING_SURFACE, 158</td>
<td></td>
</tr>
<tr>
<td>PLOGGER_CMD, 71</td>
<td></td>
</tr>
<tr>
<td>PROC_WATCH_EVENT, 383</td>
<td></td>
</tr>
<tr>
<td>PROC_WATCH_SUMMARY, 50, 383</td>
<td></td>
</tr>
<tr>
<td>PSKEEP_MODE, 180</td>
<td></td>
</tr>
</tbody>
</table>

366
PS_BUSY_COUNT, 144
PS_PENDING_BUSY, 144
PS_PENDING_LAZY, 144
PURSUIT, 225
RANGE_AVD, 204
REGION, 225
RETURN_UPDATES, 101
RETURN, 60, 259
SHADOW_CONTACT_HEADING, 221
SHADOW_CONTACT_RELEVANCE, 221
SHADOW_CONTACT_SPEED, 221
SHADOW_CONTACT_X, 221
SHADOW_CONTACT_Y, 221
STATION_KEEP, 252
SURVEY_UPDATES, 259
SURVEY, 256
TIMER_IDLE, 185
TIMER_RUNNING, 185
TRAIL_DISTANCE, 224, 226
TRAIL_HEADING, 225
TRAIL_RANGE, 225
TRAIL_RESET, 383
TRAIL_SPEED, 225
UP_LOITER, 249
USM_DECELERATION, 252
USM_DRIFT_VECTOR_ADD, 251
VIEW_MARKER, 60
VIEW_POINT, 60, 119, 123, 124, 135, 226, 235, 236, 337, 339
VIEW_POLYGON, 60, 129, 135, 150, 252, 280
VIEW_SEGLIST, 60, 76, 119, 123, 124, 204, 216
WPT_INDEX, 101, 118, 119, 123, 124
WPT_STATUS, 118
WPT_STAT, 90, 103, 119, 123, 124
Index of Configuration Parameters: All MOOS Apps

AppTick, 41, 42, 47, 49, 68
CommsTick, 42, 49, 68
Community, 42
ExtraProcessParams, 271
IVPHELM_STATE, 72
MOOSTimeWarp=1, 268
ServerHost, 42
ServerPort, 42
active_start, 71
allow_park, 65, 66
behavior_dir, 66
behaviors, 65, 66
community, 65, 66
condition, 347
deceleration, 252
domain, 65, 66
helm_prefix=F00, 65
helm_prefix, 65
hold_on_app = pBasicContactMgr, pTaskManager, 67
hold_on_app = pBasicContactMgr, 67
hold_on_app = pTaskManager, 67
hold_on_app, 65, 67
ivp_behavior_dir, 65
ok_skew, 65, 67
other_override_var, 65, 67
park_on_allstop, 64–66, 127, 261
paused, 48
post_mapping, 349
quiet, 65
standby, 73
start_in_drive=true, 64
start_in_drive, 65, 67
terse, 65
value, 42
verbose, 65, 67, 70
### Index of Behavior Parameters

<table>
<thead>
<tr>
<th>Behavior, 82</th>
<th>DEPLOY=true, 272</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAIL=true, 274</td>
<td><strong>Set, 82</strong></td>
</tr>
<tr>
<td>acomms_mark_variable, 157</td>
<td>acquire_dist, 134, 138, 140, 249</td>
</tr>
<tr>
<td>activeflag, 89, 90, 99, 114, 116, 119, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 176, 184, 186, 201, 213, 217, 222, 238, 240</td>
<td>alert_cpa_range, 280</td>
</tr>
<tr>
<td>alert_range, 278, 280</td>
<td>allowable_ttc, 149</td>
</tr>
<tr>
<td>ascent_grade, 157, 158</td>
<td>ascent_speed, 157, 158</td>
</tr>
<tr>
<td>ascent, 157</td>
<td>atsurface_status_var, 157, 158</td>
</tr>
<tr>
<td>basewidth, 143, 161, 162, 164, 165, 167, 168, 382</td>
<td>bearing_lines, 196, 202, 214, 218, 223, 240</td>
</tr>
<tr>
<td>bearing_point, 266</td>
<td>breached_altitude_flag, 128, 131</td>
</tr>
<tr>
<td>breached_depth_flag, 128, 131</td>
<td>breached_poly_flag, 128, 131</td>
</tr>
<tr>
<td>breached_time_flag, 128, 132</td>
<td>capture_delta, 170, 171</td>
</tr>
<tr>
<td>capture_flag, 170, 171</td>
<td>capture_line, 117, 121–123</td>
</tr>
<tr>
<td>capture_radius, 117, 118, 121, 123, 134, 138, 228, 229, 240, 241</td>
<td>center_activate, 134, 137, 177, 178, 182, 250, 251</td>
</tr>
<tr>
<td>center_assign, 134, 137</td>
<td>clockwise, 134, 138, 139, 141, 249</td>
</tr>
<tr>
<td>clone, 277</td>
<td>cnflag, 194, 195, 238</td>
</tr>
<tr>
<td>complete_thresh, 164, 165</td>
<td>completed_dist, 148, 203, 205, 206, 278</td>
</tr>
<tr>
<td>compression=N, 233</td>
<td>compression, 241</td>
</tr>
<tr>
<td>condition, 57, 72, 78, 80, 82, 83, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 176, 184, 186, 201, 213, 217, 222, 240</td>
<td></td>
</tr>
<tr>
<td>contact, 196, 197, 202, 214, 218, 223, 240, 279</td>
<td>convoy_flag = CONVOY_ALIGNMENT=${ALIGNMENT}, 237</td>
</tr>
<tr>
<td>cpa_flag, 148</td>
<td>crs_spd_zaic_ratio, 117</td>
</tr>
<tr>
<td>cycleflag, 117, 124, 126</td>
<td>cyleflag, 120</td>
</tr>
<tr>
<td>decay, 196, 197, 202, 214, 218, 223, 240</td>
<td>depth, 161, 170, 171, 257</td>
</tr>
<tr>
<td>depth_mismatch_var, 161</td>
<td>domain, 256</td>
</tr>
<tr>
<td>duration, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 187, 201, 213, 217, 222, 240</td>
<td>duration_reset, 98, 114, 117, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
</tr>
<tr>
<td>duration, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
<td>duration, 98, 114, 117, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
</tr>
<tr>
<td>duration_idle, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
<td>duration_idle, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
</tr>
<tr>
<td>duration_idle_decay=false, 272</td>
<td>duration_idle_decay, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
</tr>
<tr>
<td>duration_idle, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
<td>duration, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
</tr>
<tr>
<td>duration_idle, 98, 114, 116, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 240</td>
<td>edge_color, 126, 140, 183</td>
</tr>
<tr>
<td>edge_size, 126, 139, 183</td>
<td>efficiency_measure, 117</td>
</tr>
<tr>
<td>complete_dist, 148, 203, 205, 206, 278</td>
<td>endflag, 57, 78, 89, 90, 99, 114, 117, 119, 127, 133, 142, 147, 156, 160, 163, 166, 169, 173, 177, 184, 186, 201, 213, 217, 222, 238, 240</td>
</tr>
<tr>
<td>exit_on_filter_group, 193, 196, 202, 214, 218, 223, 240</td>
<td>exit_on_filter_region, 193, 196, 202, 214, 218, 223, 240</td>
</tr>
</tbody>
</table>

369
pending_status_var, 157, 158
period_busy, 143, 144
period_lazy, 143, 144
period_speed, 143
period, 157–159
perpetual=true, 124
points, 57, 82, 118, 120, 121, 259
point, 118, 120, 121
polygon = 60,-40 : 60,-160 : 150,-160 : 180,-100 : 150,-40, 147
polygon = format=radial, x=45, y=90, radius=2, pts=8, 147
polygon, 118, 128, 129, 131, 134, 136, 147, 148
poly, 148
post_mapping = CONVOY_RECAP,MY_PREFERRED_RECAP_VAR, speed_mismatch_var, 167
post_mapping = CONVOY_RECAP,silent, 237
post_mapping=WPT_INDEX,SURVEY_INDEX, 101
post_recap_verbose=true, 237
post_suffix, 118, 134, 138
post_trail_dist_on_idle, 224
priority, 57, 97, 100, 114, 117, 127, 133, 143, 148, 150, 156, 160, 163, 166, 170, 173, 177, 184, 187, 201, 206, 213, 217, 223, 240
pursueflag, 212, 215, 216
pwt_grade, 152, 203, 207
pwt_inner_dist, 148, 150, 151, 203, 206, 212, 215
pwt_inner_range, 212
pwt_outer_dist, 148, 150–152, 203, 205–207, 212, 215, 220, 224
pwt_outer_range, 212
pwt, 114, 117, 127, 133, 143, 148, 156, 160, 163, 166, 170, 173, 177, 184, 187, 201, 213, 217, 223, 240
radius, 57, 82, 118, 134, 138, 224, 226, 241
repeat, 118, 120, 124–126, 170–172, 258
reset_upon_running, 143, 145
reset_var = OPR_RESET, 132
reset_var, 128, 132
rng_flag, 148
runflag, 89, 90, 98, 114, 117, 119, 127, 133, 143, 148, 156, 160, 163, 166, 170, 173, 177, 184, 187, 201, 213, 217, 223, 238, 240
slip_radius, 118, 121, 123, 134, 138, 229, 241
slower_convoy_range=40, 231
slower_convoy_range, 231, 232, 241
spawnflag, 89, 90, 117, 128, 134, 143, 148, 156, 160, 163, 166, 170, 174, 177, 184, 187, 202, 213, 217, 223, 238, 240
spawn, 277
speed_alt, 118, 134, 137
speed_basewidth, 219, 220
templating = spawn, 147, 277
summit_delta, 143
summitdelta, 161, 162, 164, 165, 167, 168, 382
summit, 382
swing_time, 177–179, 252
tail_length_max, 228, 230, 241
templating = spawn, 147, 277
trigger_entry_time, 128, 130
trigger_exit_time, 128, 130
371
turn_range, 174, 175
updates = OBSTACLE_ALERT, 147
updates, 72, 99, 101, 106, 109, 111, 114, 117, 
128, 134, 136, 137, 143, 148, 157, 161, 
164, 167, 170, 174, 177, 182, 185, 187, 
202, 205, 213, 218, 223, 232, 233, 240, 
266, 267, 277
use_alt_speed, 118, 134, 137
use_refinery, 148, 203
vehicles_name_viewable, 249
vertex_color, 126, 139, 183
vertex_size, 126, 139, 183
visual_hint = marker_color = green, 236
visual_hint = marker_label_color = white, 236
visual_hint = marker_size = 6, 236
visual_hints, 118, 126, 128, 134, 139, 149, 178, 
182, 183, 241
wpt_index_var, 118
wpt_status_var, 118
wptflags, 123
wptflag, 118, 120, 123
xcenter_assign, 134, 137, 138
ycenter_assign, 134, 137, 138
zero_speed_depth, 157–159
capture_radius, 229
full_stop_convoy_range, 233
slip_radius, 229