MIT-CSAIL-TR-2009-028 June 18, 2009

An Overview of MOOS-IvP and a Brief Users Guide to the IvP Helm Autonomy Software

Michael R. Benjamin, Paul M. Newman, Henrik Schmidt, and John J. Leonard

An Overview of MOOS-IvP and a Brief Users Guide to the IvP Helm Autonomy Software

Michael R. Benjamin[1,2], Paul Newman[3], Henrik Schmidt[1], John J. Leonard[1]

[1]Department Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge MA

[2]Center for Advanced System Technologies
NUWC Division Newport, Newport RI

[3]Department of Engineering Science
University of Oxford, Oxford England

June 1st, 2009 - Release 4.0 beta (SVN Revision 2107)

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.moosivp.org.

This work is the product of a multi-year collaboration between the Center for Advanced System Technologies (CAST), Code 2501, of the Naval Undersea Warfare Center in Newport Rhode Island and 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
Center for Advanced System Technologies
NUWC Division Newport Rhode Island
Michael.R.Benjamin@navy.mil
mikerb@csail.mit.edu

Prof. John J. Leonard
Department of Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory
Massachusetts Intitute of Technology
jleonard@csail.mit.edu

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

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

Other collaborators have contributed greatly to the development and testing of software and ideas within, notably - Joseph Curcio, Don Eickstedt, Andrew Patrikilakis, Toby Schneider, Arjuna Balasuriya, David Battle, Christian Convey, Andrew Shafer, and Kevin Cockrell.

Sponsorship, and public release information:

This work is sponsored by Dr. Behzad Kamgar-Parsi and Dr. Don Wagner of the Office of Naval Research (ONR), Code 311. Information on Navy public release approval for this document can be obtained from the Technical Library at the Naval Undersea Warfare Center, Division Newport RI.

Contents

1 Overview

1.1 Purpose and Scope of this Document

The purpose of this document is to provide an overview of the IvP Helm in terms of design considerations, architecture and usage. This document contains references to example missions distributed with the MOOS-IvP software bundle at `www.moos-ivp.org`. The example and material herein should serve as a "getting-started" guide as well as users manual for users looking to go beyond simple autonomy missions. *THIS DOCUMENT REPRESENTS WORK IN PROGRESS. IT IS STILL CONSIDERED TO BE IN DRAFT FORM AND HAS KNOWN OMISSIONS. THE READER IS ENCOURAGED TO EMAIL THE AUTHORS FEEDBACK AND LOOK FOR LATER VERSIONS.*

1.2 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 his website. 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.

1.3 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 31 at ONR, Dr. Don Wagner and Dr. Behzad Kamgar-Parsi. 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.

1.4 The Software

The MOOS-IvP autonomy software is available at the following URL:

```
http://www.moos-ivp.org
```

Follow the links to *Software*. Instructions are provided for downloading the software from an SVN server with anonymous read-only access.

1.4.1 Building and Running the Software

After checking out the tree from the SVN server as prescribed at this link, the top level directory should have the following structure:

```
moos-ivp/
   MOOS/
   MOOS-2208/
   README.txt
   README-LINUX.txt
   README-OS-X.txt
   build-moos.sh
   build-ivp.sh
   ivp/
```

Note there is a `MOOS` directory and an `IvP` sub-directory. The `MOOS` directory is a symbolic link to a particular MOOS revision checked out from the Oxford server. In the example above this is Revision 2208 on the Oxford SVN server. This directory is left completely untouched other than giving it the local name `MOOS-2208`. The use of a symbolic link is done to greatly simplify the process of bringing in a new snapshot from the Oxford server.

The build instructions are maintained in the README files and are probably more up to date than this document can hope to remain. In short building the software amounts to two steps - building MOOS and building IvP. Building MOOS is done by executing the build-moos.sh script:

```
> cd moos-ivp
> ./build-moos.sh
```

Alternatively one can go directly into the `MOOS` directory and configure options with `ccmake` and build with `cmake`. The script is included to facilitate configuration of options to suit local use. Likewise the IvP directory can be built by executing the `build-ivp.sh` script. The `MOOS` tree must be built before building IvP. Once both trees have been built, the user's shell executable path must be augmented to include the two directories containing the new executables:

```
moos-ivp/MOOS/MOOSBin
moos-ivp/ivp/bin
```

At this point the software should be ready to run and a good way to confirm this is to run the example simulated mission in the missions directory:

```
> cd moos-ivp/ivp/missions/alpha/
> pAntler alpha.moos
```

Running the above should bring up a GUI with a simulated vehicle rendered. Clicking the `DEPLOY` button should start the vehicle on its mission. If this is not the case, some help and email contact links can be found at `www.moos-ivp.org/support/`.

1.4.2 Operating Systems Supported by MOOS and IvP

The MOOS software distributed by Oxford is well supported on Linux, Windows and Mac OS X. The software distributed by MIT/NUWC includes additional MOOS utility applications and the IvP Helm and related behaviors. These modules are support on Linux and Mac OS X. The software compiles and runs on Windows but Windows support is limited.

1.5 Where to Get Further Information

1.5.1 Websites and Email Lists

There are two websites - the MOOS website maintained by Oxford University, and the MOOS-IvP website maintained by MIT/NUWC. At the time of this writing they are at the following URLs:

```
http://www.robots.ox.ac.uk/~pnewman/TheMOOS/

http://www.moos-ivp.org
```

What is the difference in content between the two websites? 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 website is your source. For the latest on the core IvP Helm, behaviors, and MOOS tools written by MIT/NUWC, the moos-ivp.org website 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 website, 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 website 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,
```

1.5.2 Documentation

Documentation on MOOS can be found on the Oxford University website:

```
http://www.robots.ox.ac.uk/~pnewman/MOOSDocumentation/index.htm
```

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`, `pMOOSBridge`, `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.moosivp.org website under the Documentation link. Below is a summary of documents:

Documents Released or Pending Approval for Release

- *An Overview of MOOS-IvP and a Brief Users Guide to the IvP Helm Autonomy Software* (this document) - This is the primary document describing the IvP Helm regarding how it works, the motivation for its design, how it is used and configured, and example configurations and results from simulation.

- *MOOS-IvP Autonomy Tools Users Manual* - A Users Manual for seven MOOS applications: `uHelmScope`, `pMarineViewer`, `uXMS`, `uTermCommand`, `uPokeDB`, `uProcessWatch`, `pEchoVar`. These applications are common supplementary tools for running an autonomy system in simulation and on the water. See [4].

- *A Tour of MOOS-IvP Autonomy Software Modules* - This document acts as a catalog of existing modules (Both MOOS applications and IvP Behaviors). For each module, it relates (a) where it can be downloaded, (b) what the module does, (c) who it was written by, (d) rough estimate on size and complexity, and (e) what modules it may depend on for its build.

- *Extending a MOOS-IvP Autonomy System and Users Guide to the IvPBuild Toolbox* - This document is a users manual for those wishing to write their own IvP Helm behaviors and MOOS modules. It describes the IvPBehavior and CMOOSApp superclass. It also describes the IvPBuild Toolbox containing a number of tools for building IvP Functions which is the primary output of behaviors. It provides an example template directory with example IvP Behavior and MOOS application with an example CMake build structure for linking against the standard software MOOS-IvP software bundle.

Documents In-Progress

- *Extended MOOS-IvP Autonomy Examples from Simulation and In-water Exercises* - This document describes a set of example scenarios and helm configurations and describes their performance in simulation and in field exercises where possible.

- *The IvP Solver - A Look at Interval Programming as a Mathematical Programming Model* - This document describes both the mathematical structure of IvP functions and problems as well as the algorithms used for solving an IvP problem. Prior to this document being available, one can consult [5].

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).

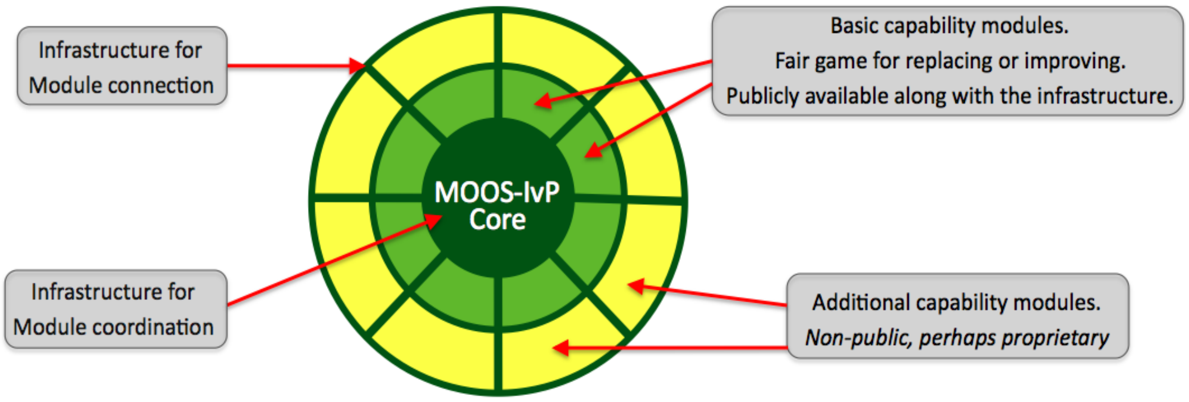

Infrastructure for
Module connection
Basic capability modules.
Fair game for replacing or improving.
Publicly available along with the infrastructure.
MOOS-IvP
Core
Infrastructure for
Module coordination
Additional capability modules.
Non-public, perhaps proprietary

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 white 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 3rd 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 to the main vehicle computer, and accepting a stream of autonomy decisions such as heading, speed and depth in return. 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.

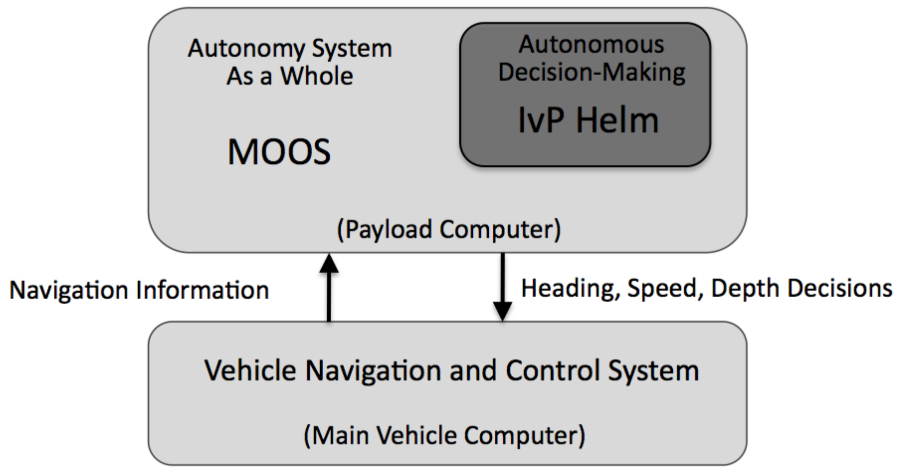


Figure 2: **The backseat driver paradigm**: The key idea is the separation of vehicle autonomy from vehicle control. The autonomy system provides heading, speed and depth commands to the vehicle control system. The vehicle control system executes the control and passes navigation information, e.g., position, 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 limited to either string or numerical values such as (`STATE, ‘‘DEPLOY’’`), or (`NAV_SPEED, 2.2`). Limiting the message type reduces the compile dependencies between modules, and facilitates debugging since all messages are human readable.

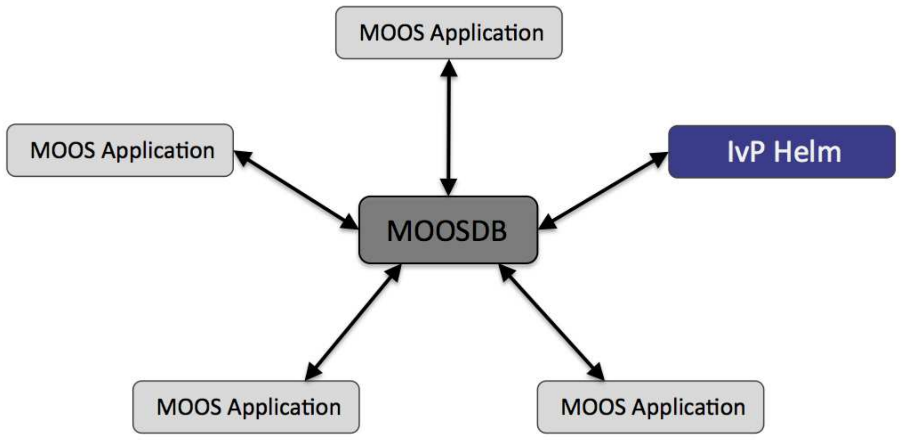


Figure 3: **A MOOS community**: is a collection of MOOS applications typically running on a single machine each with a separate process ID. Each process communicates through a single MOOS database process (the MOOSDB) in a publish-subscribe manner. Each process may be executing its inner-loop at a frequency independent from one another and set by the user. Processes may be all run on the same computer or distributed across a network.

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. 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).

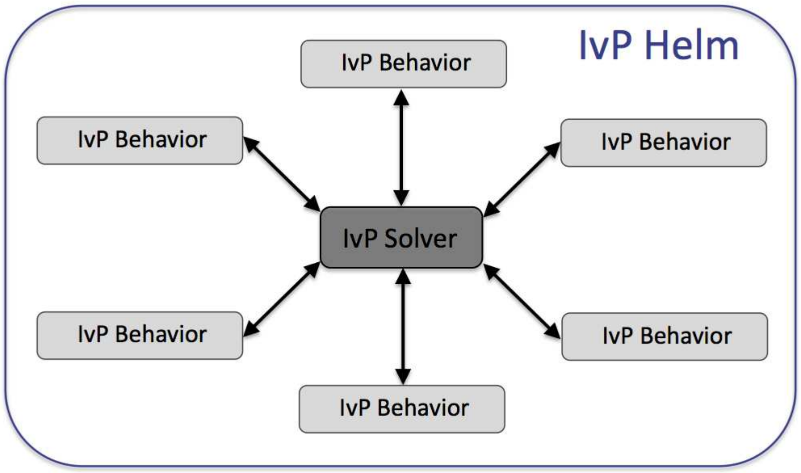


Figure 4: **The IvP Helm**: The helm is a single MOOS application running as the process pHelmIvP. It is a behavior-based architecture where the primary output of a behavior on each iteration is an IvP objective function. The IvP solver performs multi-objective optimization on the set of functions to 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 solver performs this coordination by soliciting an objective function, i.e., utility function, from each behavior defined over the vehicle decision space, e.g., possible settings for heading, speed and depth. In the IvP Helm, the objective functions are of a certain type - piecewise linearly defined - and are called IvP Functions. The solver algorithms exploit this construct to find a rapid solution to the optimization problem comprised of the weighted sum of contributing functions.

The concept of a behavior-based architecture is often attributed to [9]. 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 [9]. Another approach is referred to as the potential fields, or vector summation approach (See [1], [12]) 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], [16], [17], land vehicles, e.g., [18], and marine vehicles, e.g., [8], [10], [13], [19], [20]. However, action-selection via the identification of a single highest priority behavior and via vector summation have well known shortcomings later described in [16], [17] and [18] 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 can be argued is shorthand for the glue that connects a collection of applications where the "real" work is going on. MOOS does indeed connect a collection of applications, of which the IvP Helm is one. However, 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 extendability and reliability, it has been used in the classroom by students with no prior experience, as well 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 MOOS, see [15].

3.1 Inter-process communication with Publish/Subscribe

MOOS has a star-like topology. This is depicted in Figure 3 on page 16. Each application within a MOOS community (a MOOSApp) has a connection to a single MOOS Database (called MOOSDB) that lies 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.
- All communication between the client and server is instigated by the client, i.e., the MOOSDB never makes a unsolicited attempt to contact a MOOSApp.
- Each client has a unique name.
- A given client need have no knowledge of what other clients exist.
- A client has no way of transmitting data to a given client - it can only be sent to the MOOSDB.
- The network can be distributed over any number of machines running any combination of supported operating systems.

This centralized topology is obviously vulnerable to bottle-necking at the server regardless of how well written the server is. However the advantages of such a design are perhaps greater than its disadvantages. Firstly the network remains simple regardless of the number of participating clients. The server has complete knowledge of all active connections and can take responsibility for the allocation of communication resources. The clients operate independently with inter-connections. This prevents rogue clients (badly written or hung) from directly interfering with other clients.

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 constrained by MOOS. Somewhat unusually MOOS only allows for data to be sent in string or double form. Data is packed into messages (CMOOSMsg class) which contains other salient information shown in Table 1.

Variable	**Meaning**
Name	The name of the data
String Value	Data in string format
Double Value	Numeric double float data
Source	Name of client that sent this data to the `MOOSDB`
Time	Time at which the data was written
Data Type	Type of data (`STRING` or `DOUBLE`)
Message Type	Type of Message (usually `NOTIFICATION`)
Source Community	The community to which the source process belongs

Table 1: The contents of MOOS message

The fact that data is commonly sent in string format is often seen as a strange and inefficient aspect of MOOS. For example the string `"Type=EST,Name=AUV,Pos=[3x1]3.4,6.3,-0.23`" might describe the position estimate of a vehicle called "AUV" as a 3x1 column vector. Typically string data in MOOS is a concatenation of comma separated "name = value" pairs. It is true that using custom binary data formats does decrease the number of bytes sent. However binary data is unreadable to humans and requires structure declarations to decode it and header file dependencies are to be avoided where possible. The communications efficiency argument is not as compelling as one may initially think. The CPU cost invoked in sending a TCP/IP packet is largely independent of size up to about one thousand bytes. So it is as costly to send two bytes as it is one thousand. In this light there is basically no penalty in using strings. There is however a additional cost incurred in parsing string data which is far in excess of that incurred when simply casting binary data. Irrespective of this, experience has shown that the benefits of using strings far outweighs the difficulties. In particular:

- 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.

Of course, scalar data need not be transmitted in string format - for example the depth of a sub-sea vehicle. In this case the data would be sent while setting the data type to `"MOOS_DOUBLE"` and writing the numeric value in the double data field of the message.

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 private thread that coordinates the communication with the MOOSDB. This thread completely hides the intricacies and timings of the communications from the rest of the application and provides a small, well dened set of methods to handle data transfer. By having this thread automatically available to each MOOS application, 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 describe 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 values and double 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 is emptied the next time the `MOOSDB` accepts an incoming call from the client.

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.

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. These methods are described next.

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 uber-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 if this is the spot to write code to process the newly delivered data.

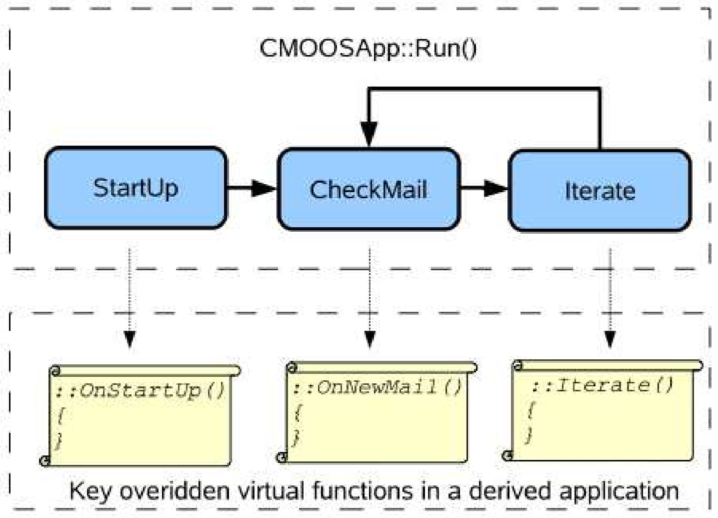


Figure 5: **Key virtual functions of the MOOS application base class**: The flow of execution once `Run()` has been called on a class derived from `CMOOSApp` . The scrolls indicate where users of the functionality of CMOOSApp will be writing new code that implements whatever it is that is wanted from the new applications.

The roles of the three virtual functions in Figure 5 are discussed below. The `pHelmIvP` application does indeed inherit from `CMOOSApp` and overload these three functions. The base class contains other virtual functions (`OnConnectToServer()` and `OnDisconnectFromServer()`) not discussed here but discussed in [15].

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 (see Section 3.5 for more on configuring an application from a 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 varCMOOSApp 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.

3.4.3 The `OnStartup()` Method

This 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 behaviour of the CMOOSApp base class) from a file. The next section (3.5) addresses the issue of configuring a MOOS application 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

```
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:

```
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 latter configures how often the communications thread talks to the `MOOSDB` and the former how often (approximately) `Iterate()` will be called. An example configuration block can be found in Listing 6 on page 42.

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-A.

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:

```
> 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

```
Run = procname [ @ LaunchConfiguration ] [ MOOSName ]
```

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 Oxford website (see Section 1.5). 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 websites, 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.
- MOOSDB http - The newer releases of MOOS allow the `MOOSDB` to be configured to run an http server on the current `MOOSDB` variable-value pairs, viewable through a web browser.

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.
- 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` 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. See Section 1.4 on page 9. 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 1 - Example uXMS *output after the* pXRelay *example is launched.*

```
 0   VarName            (S)ource       (T)ime     (C)ommunity  VarValue
 1   ----------------   ----------     ---------  ----------   ----------- (73)
 2   APPLES              n/a             n/a       n/a          n/a
 3   PEARS               n/a             n/a       n/a          n/a
 4   APPLES_ITER_HZ      pXRelay_APPLES 14.93     xrelay       24.93561
 5   PEARS_ITER_HZ       pXRelay_PEARS  14.94     xrelay       24.93683
 6   APPLES_POST_HZ      n/a             n/a       n/a          n/a
 7   PEARS_POST_HZ       n/a             n/a       n/a          n/a
```

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 [4], 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 `PEARS`) variable, which is the input variable for `pXRelay_PEARS`, by typing the following:

```
> 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 2 - Example `uPokeDB` *output after poking the MOOSDB with* `APPLES=1`*.*

```
PRIOR to Poking the MOOSDB
  VarName              (S)ource      (T)ime       VarValue
  ----------------     ----------    ----------   -------------
  APPLES


AFTER Poking the MOOSDB
  VarName              (S)ource      (T)ime       VarValue
  ----------------     ----------    ----------   -------------
  APPLES               uPokeDB       40.19        1.00000"
```

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 [4]. Once the `MOOSDB` has been poked as above, the `pXRelay_PEARS` application will receive this mail and, in return, will write to its output variable `PEARS`, which in turn will be read by `pXRelay_APPLES` 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 - Example `uXMS` *output after the* `pXRelay` *example is seeded.*

```
0   VarName           (S)ource         (T)ime    (C)ommunity   VarValue
1   ----------------   ----------      --------  ----------    ----------- (221)
2   APPLES             pXRelay_APPLES  44.78     xrelay        151
3   PEARS              pXRelay_PEARS   44.74     xrelay        151
4   APPLES_ITER_HZ     pXRelay_APPLES  44.7      xrelay        24.90495
5   PEARS_ITER_HZ      pXRelay_PEARS   44.7      xrelay        24.90427
6   APPLES_POST_HZ     pXRelay_APPLES  44.79     xrelay        8.36411
7   PEARS_POST_HZ      pXRelay_PEARS   44.74     xrelay        8.36406
```

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 `PEARS_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-A through 4-C 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 4-A - The `xrelay.moos` *mission file for the* `pXRelay` *example.*

```
 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 4-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 4-B - The `xrelay.moos` *mission file - configuring the* `pXRelay` *processes.*

```
17  //------------------------------------------
18  // pXRelay config block
```

```
19
20  ProcessConfig = pXRelay_APPLES
21  {
22    AppTick       = 25
23    CommsTick     = 25
24
25    OUTGOING_VAR  = APPLES
26    INCOMING_VAR  = PEARS
27  }
28
29  //------------------------------------------
30  // pXRelay config block
31
32  ProcessConfig = pXRelay_PEARS
33  {
34    AppTick       = 25
35    CommsTick     = 25
36
37    INCOMING_VAR  = APPLES
38    OUTGOING_VAR  = PEARS
39  }
```

In the last portion of the `xrelay.moos` file, shown in Listing 4-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 [4] for further ways to configure the `uXMS` tool.

Listing 4-C - The `xrelay.moos` *mission file for the* `pXRelay` *example - configuring* `uXMS`*.*

```
41  //------------------------------------------
42  // uXMS config block
43
44  ProcessConfig = uXMS
45  {
46    AppTick    = 4
47    CommsTick  = 4
48
49    PAUSED            = false
50    DISPLAY_SOURCE    = true
51    DISPLAY_TIME      = true
52    DISPLAY_COMMUNITY = true
53
54    VAR  = APPLES
55    VAR  = PEARS
56    VAR  = APPLES_ITER_HZ
57    VAR  = PEARS_ITER_HZ
58    VAR  = APPLES_POST_HZ
59    VAR  = PEARS_POST_HZ
60  }
```

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 `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 `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, Oxford and NUWC, and it is not even a complete list of applications from those organizations. For a more in-depth tour of MOOS applications, see [6].

3.9.1 MOOS Modules from Oxford

- `pAntler:` A tool for launching a collection of MOOS processes given a mission file. See [15], [14]. Also, see Section 3.6.
- `pMOOSBridge:` A tool that allows messages to pass between communities and allows for the renaming of messages as they are shuffled between communities. See [15], [14].
- `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 [6], [14].
- `pScheduler:` A simple tool for generating and responding to messages sent to the MOOSDB by processes in a MOOS community. See [6], [14].
- `uMS:` A GUI-Based MOOS scope for monitoring one or more MOOSDBs. See [6], [14].
- `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 [6], [14].
- `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 [6], [14].
- `iRemote:` A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associated a pre-defined variable-value poke with any un-mapped key on the keyboard. See [6], [14].

- `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 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 [6], [14].

3.9.2 MOOS Modules from MIT and NUWC

- `pHelmIvP`: The IvP Helm, and primary focus of this document.
- `pTransponderAIS`: The `pMOOSBridge` process is a tool that allows messages to pass between communities and is able to rename the messages as they are shuffled between communities.
- `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. See [4], [7]. Also, see Section 8.
- `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. See [4], [7]. Also, see Sections 3.7 and 3.8.3.
- `pMarineViewer`: A GUI-based tool primarily used for rending 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. See [4], [7]. Also, see Section 10.
- `uXMS`: A terminal based tool for live scoping on a MOOSDB process. See [4], [7]. Also, see Sections 3.7 and 3.8.2..
- `iMarineSim`: A very simple single-vehicle simulator that updates vehicle state based on present actuator values. Runs locally in the MOOS community associated with the simulated vehicle, so, unlike `uMVS`, there is one iMarineSim process running per each vehicle.
- `pEchoVar`: A lightweight process that runs without user interaction for "echoing" specified variable-value pairs posted with a follow-on post having different variable name.
- `pMarinePID`: An application providing simple PID control for vehicle speed-thrust, heading-rudder, and depth-pitch.
- `uFunctionVis`: A application for live rendering of objective functions produced by the IvP Helm behaviors. See [7].
- `uProcessWatch`: An application for monitoring the presence (connection) of a set of MOOS processes to a running `MOOSDB`. Status is summarized by a single published variable. See [4], [7].
- `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. See [4], [7].

4 A First Example with MOOS-IvP - the Alpha Mission

In this section a simple mission is described using the IvP Helm. This example is designed to run in simulation on a single desktop/laptop machine. The mission configuraiton 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 re-call the vehicle prematurely before completing the waypoints, and may subsequently command the vehicle to resume the waypoints at any time. By this example the objective is to touch 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.

4.1 Where to Find, and How to Launch the Alpha Example Mission

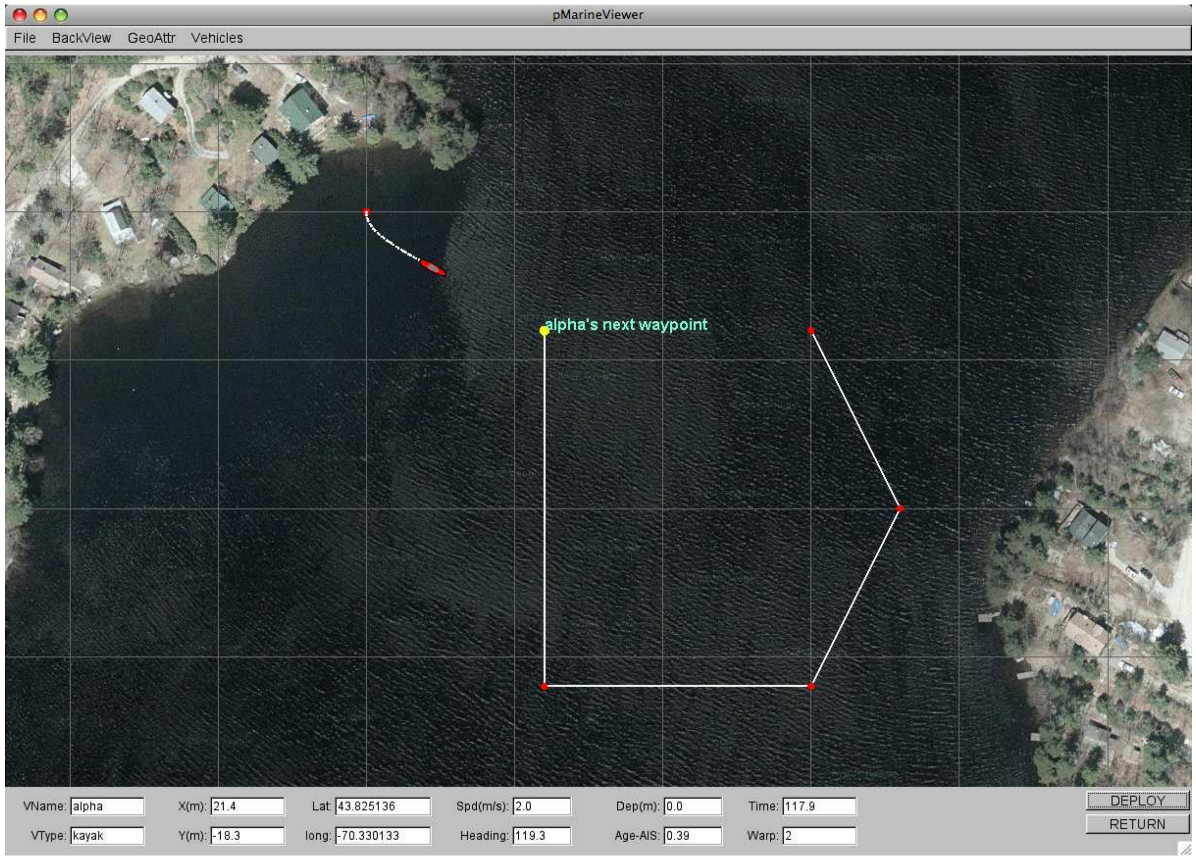
The example mission should be in the same directory tree containing the source code (See Section 1.4). There are two files - a MOOS file, also mission file or `.moos` file, and a behavior file or `.bhv` file:

```
moos-ivp/
   MOOS/
   ivp/
      missions/
         alpha/
            alpha.moos    <---- The MOOS file
            alpha.bhv     <---- The Behavior file
```

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

```
> cd moos-ivp/ivp/missions/alpha
> pAntler alpha.moos
```

After `pAntler` has launched each process, the `pMarineViewer` window should be open and look similar to that shown in Figure 6. After clicking the `DEPLOY` button in the lower right corner the vehicle should start to traverse the shown set of waypoints.

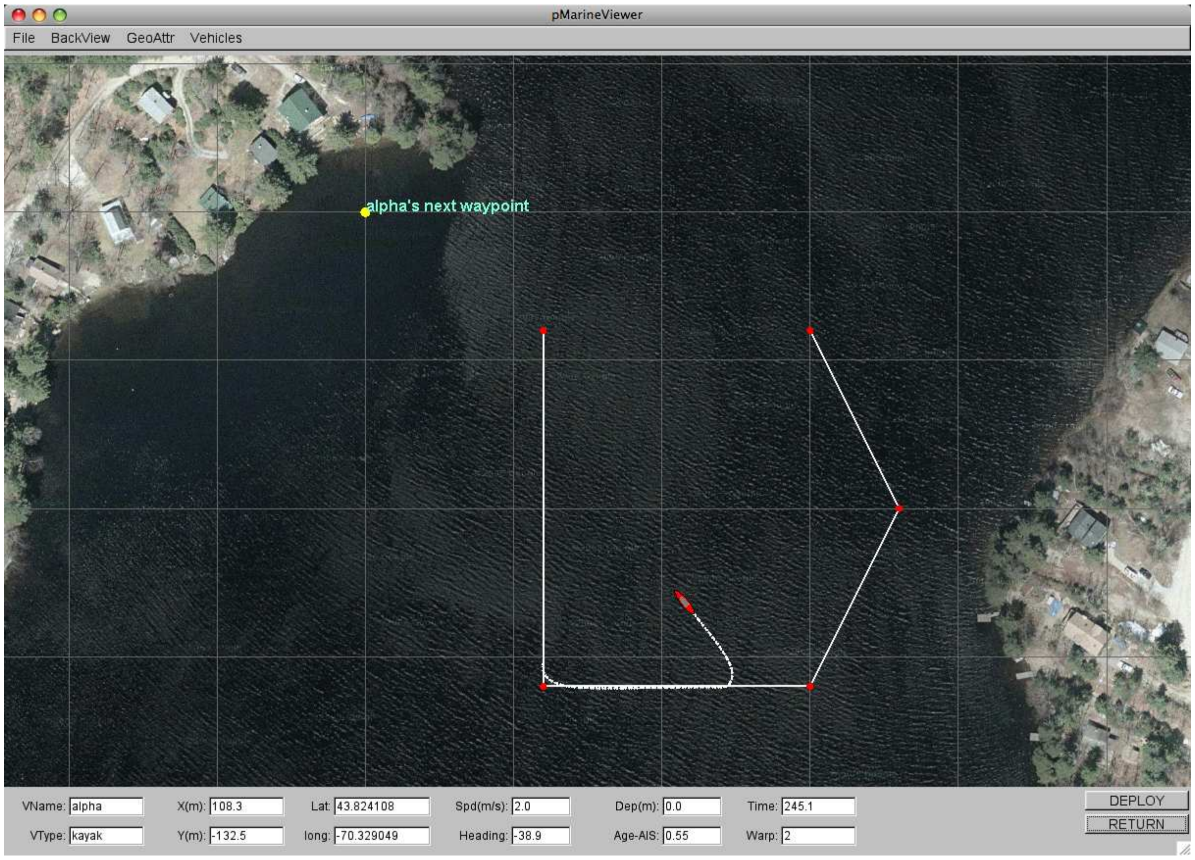


VName: alpha X(m): 21.4 Lat: 43.825136 Spd(m/s): 2.0 Dep(m): 0.0 Time: 117.9

VType: kayak Y(m): -18.3 long: -70.330133 Heading: 119.3 Age-AIS: 0.39 Warp: 2

Figure 6: **The Alpha Example Mission - In the Surveying Mode**': A single vehicle is dispatched to traverse a set of waypoints and, upon completion, traverse to the waypoint (0,0) which is the launch point.

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 7. When the vehicle is returning as in the figure, it can be re-deployed by hitting the `DEPLOY` button again.



VName: alpha
VType: kayak
X(m): 108.3
Y(m): -132.5
Lat: 43.824108
long: -70.329049
Spd(m/s): 2.0
Heading: -38.9
Dep(m): 0.0
Age-AIS: 0.55
Time: 245.1
Warp: 2

Figure 7: **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 Closer 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 5. 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 5: The behavior file for the Alpha example.

```
0 //--------    FILE: alpha.bhv   -------------
1
2 initialize   DEPLOY = false
3 initialize   RETURN = false
4
5 //----------------------------------------------
6 Behavior = BHV_Waypoint
7 {
 8   name      = bhv_waypt_survey
 9   priority  = 100
10   condition = DEPLOY = true
11   condition = RETURN = false
12   endflag   = RETURN = true
13
14   speed     = 2.0
15   radius    = 8.0
16   points    = 60,-40:60,-160:150,-160:180,-100:150,-40
17 }
18
19 //----------------------------------------------
20 Behavior = BHV_Waypoint
21 {
22   name      = bhv_waypt_return     // IvPBehavior common param
23   priority  = 100                  // IvPBehavior common param
24   condition = RETURN = true        // IvPBehavior common param
25   condition = DEPLOY = true        // IvPBehavior common param
26
27   speed     = 2.0               // BHV_Waypoint specific param
28   radius    = 8.0               // BHV_Waypoint specific param
29   points    = 0,0               // BHV_Waypoint specific param
30 }
```

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 20-30) 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 10-11, 24-25), and endflags (line 12) are primary tools for coordinating separate behaviors into a particular mission. Behaviors will not participate unless each of its conditions are met. The condtions 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 2, and is toggled by the `DEPLOY` button on the `pMarineViewer` GUI shown in Figures 6 and 7. The `pMarineViewer` MOOS application is but one example of a command and control interface to the helm. The MOOS variables in the behavior conditions in Listing 5 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 24) met when the surveying behavior posts its endflag on line 12.

4.3 A Closer Look at the set of 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 8.

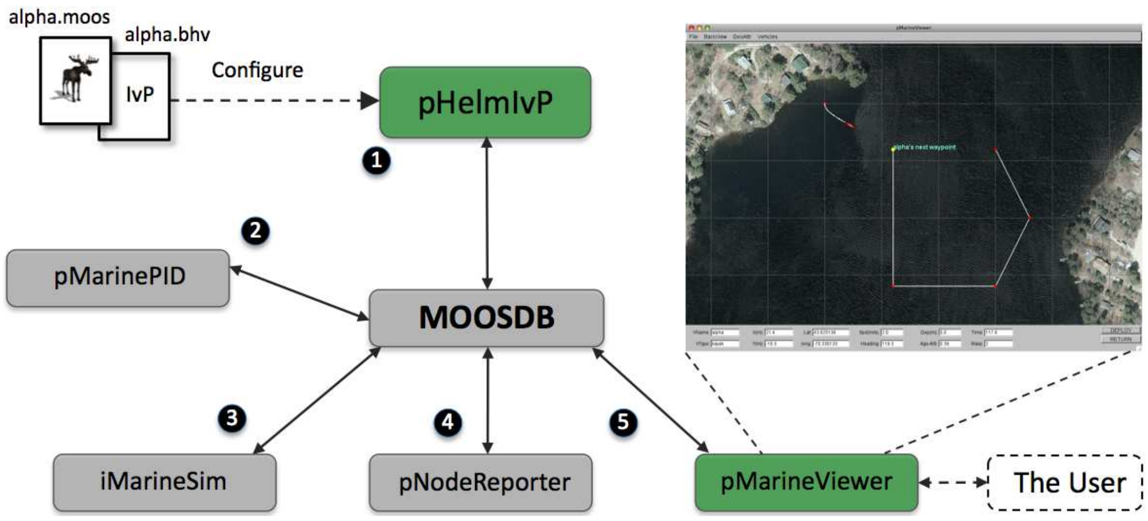


Figure 8: **The MOOS processes in the example "alpha" mission**: In (1) The helm produces a desired heading and speed. In (2) the PID controller subscribes for the desired heading and speed and publishes actuation values. In (3) the simulator grabs the actuator values and the current vehicle pose and 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, the viewer. In (5) the pMarineViewer grabs the node-report and renders a new vehicle posision. The user can interact with the viewer to write limited commmand and control variables to the MOOSDB.

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 procoess a pointer to this same MOOS file. The Antler configuration block in this example looks like

```
ProcessConfig = ANTLER
{
  MSBetweenLaunches = 200

  Run = MOOSDB          @ NewConsole = false
  Run = iMarineSim      @ NewConsole = false
  Run = pNodeReporter   @ NewConsole = false
  Run = pMarinePID      @ NewConsole = false
  Run = pMarineViewer   @ NewConsole = false
  Run = pHelmIvP        @ NewConsole = false
}
```

The first parameter specifies how much time should be left beteen the launching of each process. The other lines 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. You might try switching one or more of these to `true` as an experiment.

4.3.2 The pMarinePID Application and Configuration Block

The `pMarinePID` application implements a simple PID controller which produces values suitable for actuator control based on inputs from the helm. The full configuration for this block can be found in the appendix. In simulation the output is consumed by the vehicle simulator rather than the vehicle actuatiors.

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

Subcribes to: `DESIRED_HEADING`, `DESIRED_SPEED`.

Publishes to: `DESIRED_RUDDER`, `DESIRED_THRUST`.

4.3.3 The iMarineSim Application and Configuration Block

The `iMarineSim` 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:

```
ProcessConfig = iMarineSim
{
  AppTick = 10
  CommsTick = 10

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

In short: The `iMarineSim` application typically gets its info from `pMarinePID`; produces info consumed by `pNodeReporter` and itself on the next iteration of `iMarineSim`.

Subcribes 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 equiped with an AIS receiver. It periodically collects all latest pose elements, e.g., latitutude 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 `iMarineSim` or otherwise on-board navigation systms such as GPS or compass; produces info consumed by `pMarineViewer` and instances of `pHelmIvP` running in other vehicles or simulated vehicles.

Subcribes 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 `AIS_REPORT_LOCAL` 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).

Subcribes 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 the larger MOOS community.

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 9. 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.

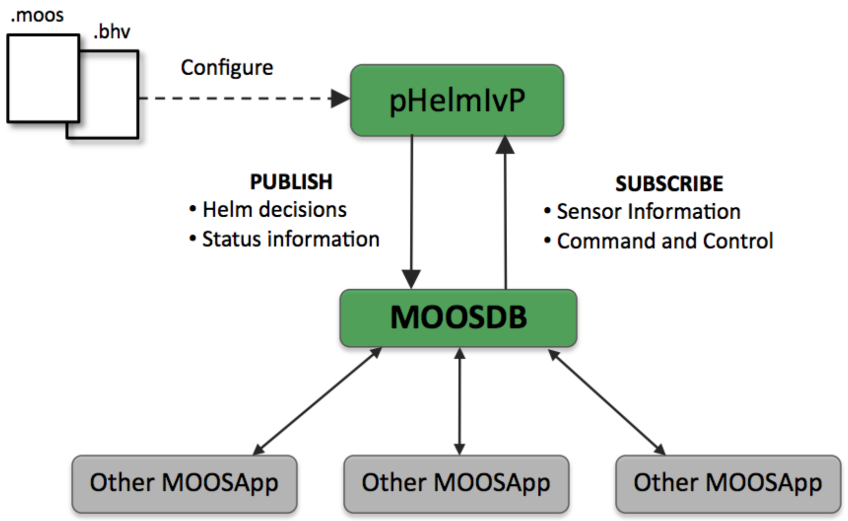


Figure 9: **The `pHelmIvP` MOOS application**: The IvP Helm is implemented as the MOOS application `pHelmIvP`. The helm is configured with two files - the mission file and behavior file. Once launched it connects to the MOOSDB along with other MOOS applications performing other functions. Information flowing into the helm include both sensor information and command and control inputs. The helm produces commands for maneuvering the vehicle along with other status information produced by active behaviors.

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.

5.2 Helm Engagement

The highest level interface with the helm concerns simply whether it is *engaged* or *disengaged*. To use an automobile analogy, launching the `pHelmIvP` MOOS process is like turning the car on, and putting the helm in the engaged mode is like shifting from "Park" to "Drive". Here we discuss (a) how the engagement is changed, (b) what it is going on in the helm when it is disengaged, (c) how the helm engagement state is initialized at start-up.

5.2.1 Helm Engagement Transitions

The helm engagement state can be transitioned by writing to the MOOS variable `MOOS_MANUAL_OVERIDE`. As Figure 10 depicts, a value of `false`, which is case insensitive, puts the helm in the `Engaged` state. A value of `true` puts it into the `Disengaged` state. When the helm transitions from `Engaged` to `Disengaged` it makes *one* more publication to the helm decision variables, each with a value zero. This can be thought of publishing "All-Stop".

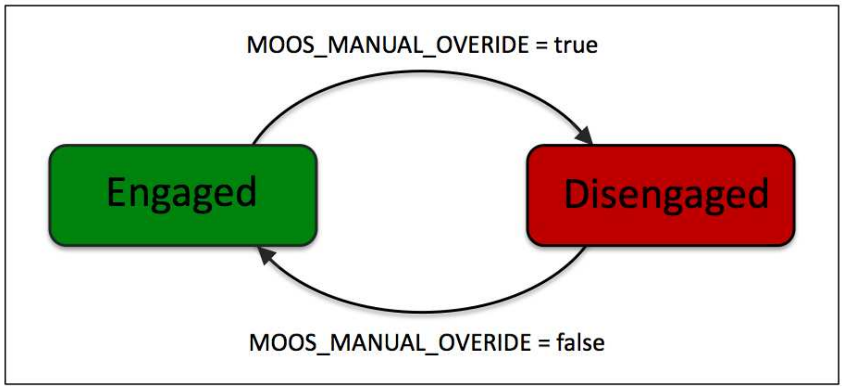


Figure 10: **The Engagement state of the IvP Helm**: The helm is either engaged or disengaged, depending on both how the helm is initialized and mail received by the helm after start-up on the variable `MOOS_MANUAL_OVERIDE.` The value for this variable is case insensitive.

The variable `MOOS_MANUAL_OVERIDE` 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 engagement state by opening a scope on the MOOSDB, on either the wrong variable or the two disagreeing variables. In this case the helm engagement state would be aligned with the variable with the most recent publication time stamp. In any event, the best way to monitor the helm engagement state is to scope on the MOOS variable `HELM_ENGAGEMENT`, published by the helm itself, or use the `uHelmScope` tool.

The helm can also be transitioned interally from the `Engaged` to the `Disengaged` state if a behavior determines that a critical error has occurred, such as not getting critical sensor information for a critical safety behavior. In this situation, the helm can be re-engaged by another MOOS client posting `MOOS_MANUAL_OVERIDE=false`, but this is no guarantee that the helm may disengage again immediately if the same condition persists that caused a behavior to declare a critical error previously.

5.2.2 What Is and Isn't Happening when the Helm is Disengaged

When the helm is in the `Disengaged` state, the loop depicted in Figure 5 on page 21 carries on. The `OnNewMail()` continues to be called and new mail is read and dealt with exactly as it would if the helm were in the `Engaged` state. 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. 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 Engagment State at Process Launch Time

The helm, by default, is configured to be initiallly in the `Disengaged` state upon start-up. By setting parameter `START_ENGAGED=true` in the mission file configuration block, the helm will indeed be in the `Engaged` state 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, engaged and ready to accept field commands. This feature should be used with caution, and it may be phased out in a later software release.

5.2.4 Suggestions for Trying Out the Engagement Settings

- Try running the Alpha mission again from Section 4. Deploy the vehicle, and open a separate console from which to poke the MOOSDB with the following:

  ```
  > uPokeDB alpha.moos MOOS_MANUAL_OVERIDE=true
  ```

  which should pause the vehicle in its track. Then resume the vehicle with another poke, this time with `MOOS_MANUAL_OVERIDE=false`.

- Try running the Alpha mission again, and as above, toggle the value of `MOOS_MANUAL_OVERIDE`. This time however, open a separate console and run `uHelmScope` by typing:

  ```
  > uHelmScope alpha.moos MOOS_MANUAL_OVERIDE IVPHELM_ENGAGED "
  ```

  Note the appearance of `"DISENGAGED!!!"` in the top line of the output whenever the helm is in the `Disengaged` state. Also note the values of `MOOS_MANUAL_OVERIDE` and `IVPHELM_ENGAGED` in the MOOSDB Scope section of the `uHelmScope` output.

5.3 Parameters for the `pHelmIvP` MOOS Configuration Block

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

Parameter	Mandatory	Description
`COMMUNITY`	YES	Global MOOS parameter. Determines ownship name
`DOMAIN`	YES	The decision space for the IvP Solver
`BEHAVIORS`	NO	The name and location of the behavior configuration file
`START_ENGAGED`	NO	Determines whether or not the helm is override mode at start-up.
`VERBOSE`	NO	Determines verbosity of terminal output - `quiet`, `terse`, or `verbose`
`OK_SKEW`	NO	Tolerance on the age of incoming mail before rejected as being too old

Table 2: Configuration parameters for the `pHelmIvP` block in a typical MOOS mission configuration file.

5.3.1 The `COMMUNITY` Parameter

The `COMMUNITY` 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.

5.3.2 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 6 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`.

5.3.3 The `BEHAVIORS` Parameter

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 an optional parameter because a file could alternatively be specified on the command line (when not launching with pAntler). If a behavior file is specified both on the command line, and in the `pHelmIvP` configuration block with this parameter, they will all be used to configure the helm behaviors collectively.

5.3.4 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.

5.3.5 The START_ENGAGED Parameter

This is an optional parameter. This parameter is set to either `true` or `false`. Normally the helm starts in the `Disengaged` state and needs to receive MOOS mail on the variable `MOOS_MANUAL_OVERIDE` with the value of this variable set to `true`. When `START_ENGAGED` is set to `true`, the helm is in the `Engaged` state upon start-up. The issue of helm engagement was discussed in more detail in Section 5.2.

5.3.6 The OK_SKEW Parameter

This is an optional parameter. 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 skews can be disabled by setting `OK_SKEW = ANY`. The default value is 60 seconds.

5.3.7 An Example pHelmIvP MOOS Configuration Block

Below is an example configuration block for the IvP Helm.

Listing 6 - An example `pHelmIvP` *configuration block.*

```
//-------- pHelmIvP configuration block  -------------
ProcessConfig = pHelmIvP
{
  AppTick    = 4   // Defined for all MOOS processes
  CommsTick  = 4   // Defined for all MOOS processes

  Domain     = course:0:359:360
  Domain     = speed:0:3:16
  Domain     = depth:0:500:101

  // IF BELOW COMMENTED OUT, FILE GIVEN ON CMD-LINE
  Behaviors = foobar.bhv

  // VERSBOSE = terse produces minimal terminal output
  VERBOSE = terse

  // ACTIVE_START = false is the default
  START_ENGAGED = true

  // OK_SKEW = 60 (seconds) is the default
  OK_SKEW = ANY
}
```

The `APPTICK` and `COMMSTICK` parameters are defined for all MOOS processes (see [15]) 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.

5.4 Launching the IvP Helm and Output to the Terminal Window

The IvP 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 paramters.
[-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 7 below.

Listing 7 - Example start-up output generated by the `pHelmIvP` *process.*

```
***************************************************
*                                                 *
*       This is MOOS Client                       *
*       c. P Newman 2001                          *
*                                                 *
***************************************************

---------------MOOS CONNECT-----------------------
  contacting a MOOS server localhost:9000 -  try 00001
  Contact Made
  Handshaking as "pHelmIvP"
  Handshaking Complete
  Invoking User OnConnect() callback...ok
--------------------------------------------------

The IvP Helm (pHelmIvP) is starting....
Loading behavior dynamic libraries....
    Loading directory: /Users/mikerb/project-colregs/src/lib_behaviors-colregs
Loading behavior dynamic libraries - FINISHED.
Number of behavior files: 1
Processing Behavior File: bravo.bhv  START
    Successfully found file: bravo.bhv
    InitializeBehavior: found static behavior BHV_Loiter
    InitializeBehavior: found static behavior BHV_Loiter
    InitializeBehavior: found static behavior BHV_Waypoint
    InitializeBehavior: found static behavior BHV_Timer
Processing Behavior File: bravo.bhv  END
pHelmIvP is Running:
  AppTick   @ 4.0 Hz
  CommsTick @ 4 Hz
```

```
30    Time Warp @ 1.0
31  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

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 IvP 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 7. 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 7, a series of dollar-signs, 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 8:

Listing 8 - An example helm iteration report generated by an active helm.

```
 0  Iteration: 161  *****************************************
 1  Helm Summary  ---------------------------
 2  loiter_a did NOT produce an obj-function
 3  loiter_b produces obj-function - time:0.00 pcs: 9.00000 pwt: 100.00000
 4  waypt_return did NOT produce an obj-function
 5  loiter_timer did NOT produce an obj-function
 6  Number of Objective Functions: 1
 7  DESIRED_SPEED:  2.10
 8  DESIRED_COURSE:  145.00
 9  (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. This tool is described later in Section 8.

5.5 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.

5.5.1 Variables published by the IvP Helm

Variables published by the IvP Helm are summarized in Table 3 below. The column indicating frequency is in respect to each helm iteration. A more detailed description of each variable follows the table.

#	Variable	Freq	Description
1	IVPHELM_SUMMARY	Each	Summary of many statistics of the current helm iteration, current mode.
2	IVPHELM_POSTINGS	Each	Recap of all variable-value behavior posting for the current iteration.
3	IVPHELM_STATEVARS	Rare	List of variables involved in behavior preconditions.
4	IVPHELM_MODESET	Once	Description of Helm Hierarchical Mode Declarations.
5	IVPHELM_ENGAGED	Rare	Status of the Helm Engagement State (true or false).
6	HELM_IPF_COUNT	Each	IvP Functions involved in the decision of the most recent iteration.
7	CREATE_CPU	Each	Total time needed to create IvP Functions in the most recent iteration.
8	LOOP_CPU	Each	Total time in the Iterate() loop of the most recent iteration.
9	PLOGGER_CMD	Once	A hook to the pLogger to record the behavior file(s).
10	DESIRED_*	Most	The result of the Helm in its configured decision space.
11	BHV_IPF	Most	String form of IvP functions produced by behaviors.
12	BHV_WARNING	Rare	Warning messages generated by helm behaviors.
13	BHV_ERROR	Rare	Error messages generated by helm behaviors.

Table 3: Variables published by the IvP Helm.

- IVPHELM_SUMMARY: Produced on each iteration of the helm for consumption by the uHelmScope application. 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.
- IVPHELM_POSTINGS: Produced on each iteration of the helm for consumption by the uHelmScope application. It provides a recap of all variable-value postings made by all behaviors on the

current iteration.

- `IVPHELM_STATEVARS`: Produced periodically by the helm for consumption by the `uHelmScope` application. 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 (see Section 8.) It contains the specification of the Hierarchical Mode Declarations, if any, in use by the helm.
- `IVPHELM_ENGAGED`: Written by the helm once each time it changes the Engagement State (see Section 5.2). It is either `true` or `false`.
- `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.

5.5.2 Variables Subscribed for by the IvP Helm

Variables subscribed for by the IvP Helm are summarized in Table 4 below. A more detailed description of each variable follows the table.

#	Variable	Description
1	MOOS_MANUAL_OVERIDE	Allows for transition of the helm Engagement State.
2	MOOS_MANUAL_OVERRIDE	Allows for transition of the helm Engagement State.
3	HELM_MAP_CLEAR	Resets the helm map that filters successive duplcate publications.

Table 4: Variables subscribed for by the IvP Helm.

- `MOOS_MANUAL_OVERIDE`: 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 5.4.
- `HELM_MAP_CLEAR`: When received, the helm clears an internal map that is used to surpress repeated duplicate postings. See Section 5.6.

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.

5.6 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 noticably facilitate the use of post-mission alalysis 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 discusss it here to explain a bit what is behind the variable `HELM_MAP_CLEAR` to which the helm subscribes, and listed above in Section 5.5.2.

5.6.1 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. For sure, 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. For example, a waypoint behavior named

"SURVEY" will post, on each helm iteration, the variables PWT_BHV_SURVEY and STATE_BHV_SURVEY indicating the behavior's priority and run-state. These variable values often remain unchanged for many successive iterations, and really only need to be posted upon a change.

The uHelmScope tool depends on a number of status variables published by the helm to provide content for the scope. These variables are the IVPHELM_* variables listed in Table 3. This includes the variable IVPHELM_POSTINGS which is a summary of *all* variable-value postings made by *all* behaviors on the current iteration. This provides the content for the Behavior-Posts section of the uHelmScope output, described in Section 8.2.3 on page 85. This string can be long, and the point here is that each unnecessary successive duplicate post by a behavior actually shows up in the log file twice! They can also clutter the output in the uHelmScope window, but main detriment motivating the filter is the reduction of log file bloat.

5.6.2 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 two types of message types in MOOS (see Section 3.2 on page 18). 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 ommitted 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.

5.6.3 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, [9]. 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, [11]. 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 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 [16] and [18] - 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 5 on page 33.

6.2 Inside the IvP 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 11. Description of the five components follow.

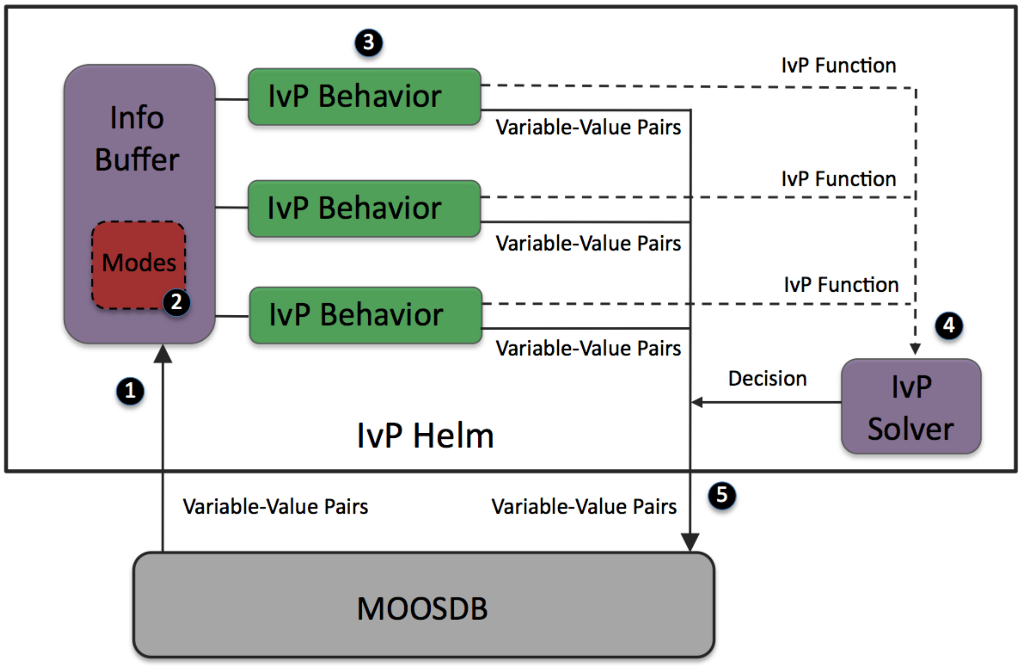


Figure 11: **The pHelmIvP Iterate Loop**: (1) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors, (2) If there were any mode declarations in the mission behavior file they are evaluated at this step. (3) Each behavior is queried for its contribution 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 on page 21, 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 11 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 11, 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 5.6, 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 5 on page 33 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>
```

One initialization per line. 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 variable `<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 11 on page 52, 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.

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>
{
  <parameter> = <value>
  ...
  <parameter> = <value>
}
```

The first line is a declaration of the behavior type. The keyword `Behavior` is not case sensitive, but the `<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 `<parameter>` and `<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 5 on page 33, the general behavior parameters are listed on lines 8-12 and 22-25, but the parameters specific to the waypoint behavior, `speed`, `radius`, and `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 `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:

```
Set <mode-variable-name> = <mode-value>
{
  <mode-variable-name> = <parent-value>
  <condition>
  . . .
  <condition>
} <else-value>
```

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

As indicated in Figure 11, the value of each mode variable is reset at the outset of the `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 5. 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.

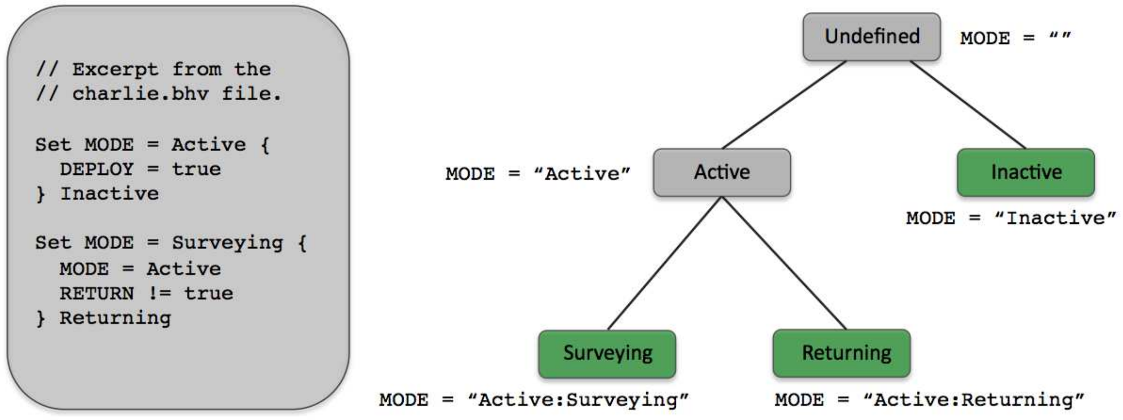
`DEPLOY`	`RETURN`	Mode
`true`	`true`	Returning
`true`	`false`	Surveying
`false`	`true`	Idle
`false`	`false`	Idle

Table 5: Possible modes implied by the condition parameters in the alpha mission in Listing 5.

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 5 really should read `RETURN != true`, and not `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 5 would not be satisfied, and the vehicle would be in the idle state, despite the fact that `DEPLOY` may be set to `true`. These problems are alleviated by the use of hierarchical mode declarations.

6.4.3 Syntax of Hierarchical Mode Declarations - The Charlie Mission

We provide an example of the use of hierarchical mode declarations by extending the Alpha mission described in Section 4. This example mission is dubbed the "Charlie" mission. The `charlie.bhv` file can be found alongside the alpha mission in the MOOS-IvP distribution (Section 4.1). It is also given fully in Listing 9 on the next page. The *implicit* modes of the Alpha mission, described in Table 5, are explicitly declared in the Charlie behavior file to form the following hierarchy:



```
// Excerpt from the
// charlie.bhv file.

Set MODE = Active {
  DEPLOY = true
} Inactive

Set MODE = Surveying {
  MODE = Active
  RETURN != true
} Returning
```

Figure 12: **Hierarchical modes for the Charlie mission**: The vehicle will always be in one of the modes represented by a leaf node. A behavior may be associated with any node in the tree. If a behavior is associated with an internal node, it is also associated with all its children.

The hierarchy in Figure 12 is formed by the mode declaration constructs on the left-hand side, taken as an excerpt from the `charlie.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 12 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 12 `MODE` will be set to either `"Active"` or `"Inactive"`. When the second block is evaluated, the condition `"MODE=Active"` is evaluate base 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.

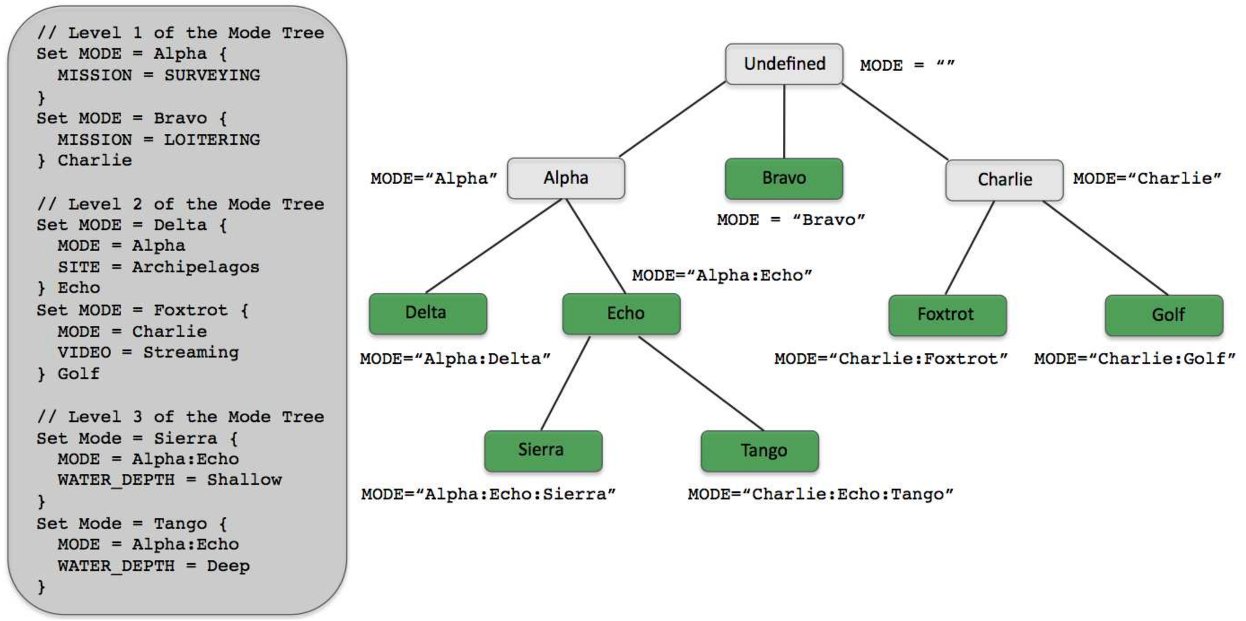
Once the mode is evaluated, at the outset of the helm iteration, it is available for use in the conditions of the behaviors, as in lines 20 and 23 in Listing 9. Note the `"=="` relation in lines 20 and 23. This is a string-matching relation that matches when one side matches exactly one of the components 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.

Listing 9: The Charlie Mission - Use of Hierarchical Mode Declarations.

```
//--------    FILE: charlie.bhv   -------------

initialize   DEPLOY = false
initialize   RETURN = false

//------------------- Declaration of Hierarchical Modes
set MODE = ACTIVE {
  DEPLOY = true
} INACTIVE

set MODE = SURVEYING {
  MODE = ACTIVE
  RETURN != true
} RETURNING

//---------------------------------------------
Behavior = BHV_Waypoint
{
  name      = waypt_survey
  pwt       = 100
  condition = MODE == SURVEYING
  endflag   = RETURN = true

  speed     = 2.0   // meters per second
  radius    = 8.0
  points    = 60,-40:60,-160:150,-160:180,-100:150,-40
}

//---------------------------------------------
Behavior = BHV_Waypoint

  name       = waypt_return
  pwt        = 100
  condition  = MODE == RETURNING
  updates    = UPDATES_RETURN

  speed      = 2.0
  radius     = 8.0
  points     = 0,0
}
```

6.4.4 A More Complex Example of Hierarchical Mode Declarations

The Charlie 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 13. The declarations are given in the order of layers of the tree ensuring that parents are declared prior to children. As with the Charlie example in Figure 12, the nodes that represent realizable modes are depicted in the darker (green) color.



```
// Level 1 of the Mode Tree
Set MODE = Alpha {
  MISSION = SURVEYING
}
Set MODE = Bravo {
  MISSION = LOITERING
} Charlie

// Level 2 of the Mode Tree
Set MODE = Delta {
  MODE = Alpha
  SITE = Archipelagos
} Echo
Set MODE = Foxtrot {
  MODE = Charlie
  VIDEO = Streaming
} Golf

// Level 3 of the Mode Tree
Set Mode = Sierra {
  MODE = Alpha:Echo
  WATER_DEPTH = Shallow
}
Set Mode = Tango {
  MODE = Alpha:Echo
  WATER_DEPTH = Deep
}
```

Figure 13: **Example Hierarchical Mode Declaration**: The hierarchy on the right is constructed from the set of mode declarations on the 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 (see Section 8.2.2), but it is also displayed on it's own, in the fourth line of the main output. For example, see line 4 in Listing 14 on page 83. Unlike the other general MOOS scope tools, the `uHelmScope` allows for stepping backwards through helm iterations to see when the mission mode changed and perhaps what precipitated the change. See the section on stepping through saved scope history in Section 8.3.

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 13. The `uHelmScope` output for the example in Figure 13 is shown in listing 10 below.

Listing 10: The mode hierarchy output from `uHelmScope` *for the example in Figure 13.*

```
ModeSet Hierarchy:
---------------------------------------------
Alpha
    Delta
    Echo
        Sierra
        Tango
Bravo
Charlie
    Foxtrot
    Golf
---------------------------------------------
CURENT MODE(S): Charlie:Foxtrot

Hit 'r' to resume outputs, or SPACEBAR for a single update
```

More on this feature of the `uHelmScope` can be found in Section 8. 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 11 on page 52, 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>
```

Each logic expression is comprised of either Boolean operators (and, or, not) or relation operators ($\leq, <, \geq, >, =, !=$). All expressions have at least one relational expression, where the left-hand side of the expression is treated as a 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:

```
DEPLOY   = true    // Example 1
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 the MOOS variable `DEPLOY` had the value `"TRUE"`, this would satisfy the condition in Example 1 above. But if the MOOS variable `deploy` had the value `"true"`, this would not satisfy Example 1. 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:

```
(DEPLOY = true) or (QUALITY >= 75)             // Example 3

(MSG != error) and !((K <= 10) or (w != 0))     // Example 4
```

A relational expression such as `(w != 0)` above is false if the variable `w` is undefined. In MOOS, this occurs if 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 3 would evaluate to false even if the variable `w` had the string value `"alpha"` which is clearly not equal to zero.

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:

```
REQUESTED_STATE != $(RUN_STATE)               // Example 5
```

The variable types need to match or the expression will evaluate to false regardless of the relation. The expression in Example 5 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.

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 Charlie mission in Listing 9 on page 58, as an alternative to their usage in the Alpha mission example in Listing 5 on page 33.

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 13:

```
"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 14:

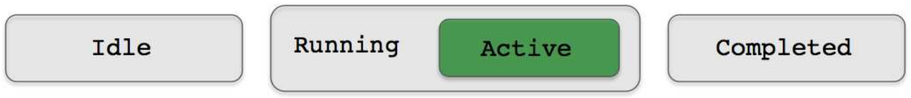


Figure 14: **Behavior States:** A behavior may be in one of these four states at any given iteration of helm `Iterate()` loop. The state is determined by examination of MOOS variables stored locally in the helm's information buffer.

- 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 11, p. 52). 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 5.6.

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 five types, (1) `endflag`, (2) `idleflag`, (3) `runflag`, (4) `activeflag`, (5) `inactiveflag`. The variable-value pairs representing each flag are set in the behavior file for the corresponding behavior. See line 12 in 5 on page 33 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 on each iteration of the helm when the behavior is determined to be in 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`: An `runflag` is posted on each iteration of the helm when the behavior is determined to be in the `running` state, regardless of whether it is further determined to be active or not. 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 on each iteration of the helm when the behavior is determined to be in 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 on each iteration of the helm when the behavior is determined to be not in 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.

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 15:

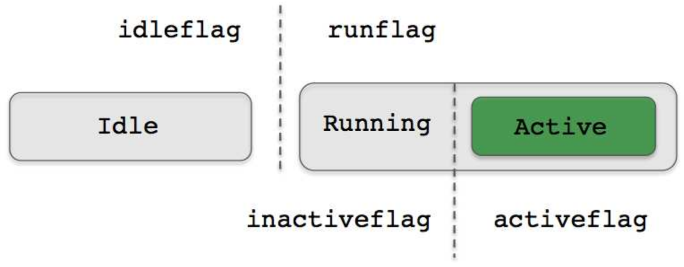


Figure 15: **Behavior Flags:** The four behavior flags `idleflag`, `runflag`, `activeflag`, and `inactiveflag` are posted depending on the behavior state and can be considered complementary in the manner indicated.

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, as in lines 12-17 in Listing 14 on page 83. These lines are provided 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 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.

The uHelmScope tool can also be used to monitor the messages generated by each behavior on each iteration. The helm, in addition to posting all the variable-value pairs to the MOOSDB at the end of the Iterate() loop, also builds a summary of all such posts into a single string and publishes it as IVPHELM_POSTINGS. This variable is subscribed for and parsed by uHelmScope to generate the "Behavior-Posts" section of the uHelmScope output. An example can be seen in lines 28-39 in Listing 14, and this part of the uHelmScope output is described in Section 8.2.3.

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 11, p. 52). 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 5.3.2, p. 41). 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 17).
- 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 16:

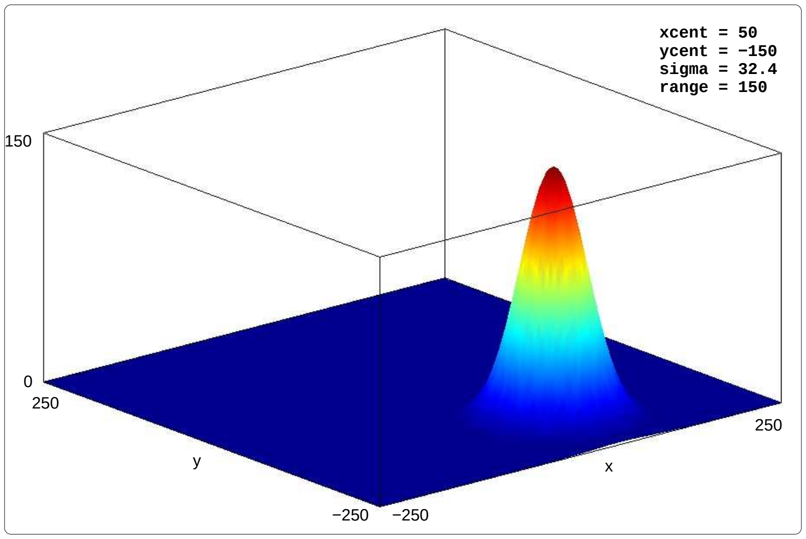


Figure 16: A rendering of the function $f(x, y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where $A =$ `range` $= 150$, $\sigma =$ `sigma` $= 32.4$, $x_0 =$ `xcent` $= 50$, $y_0 =$ `ycent` $= -150$. The domain here for x and y ranges from -250 to 250.

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 17(a) and 17(b). The difference in these two figures is only the size of the piece. More pieces (Figure 17(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.

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 rendered in Figure 17(c). 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 rendered in Figure 17(d).

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

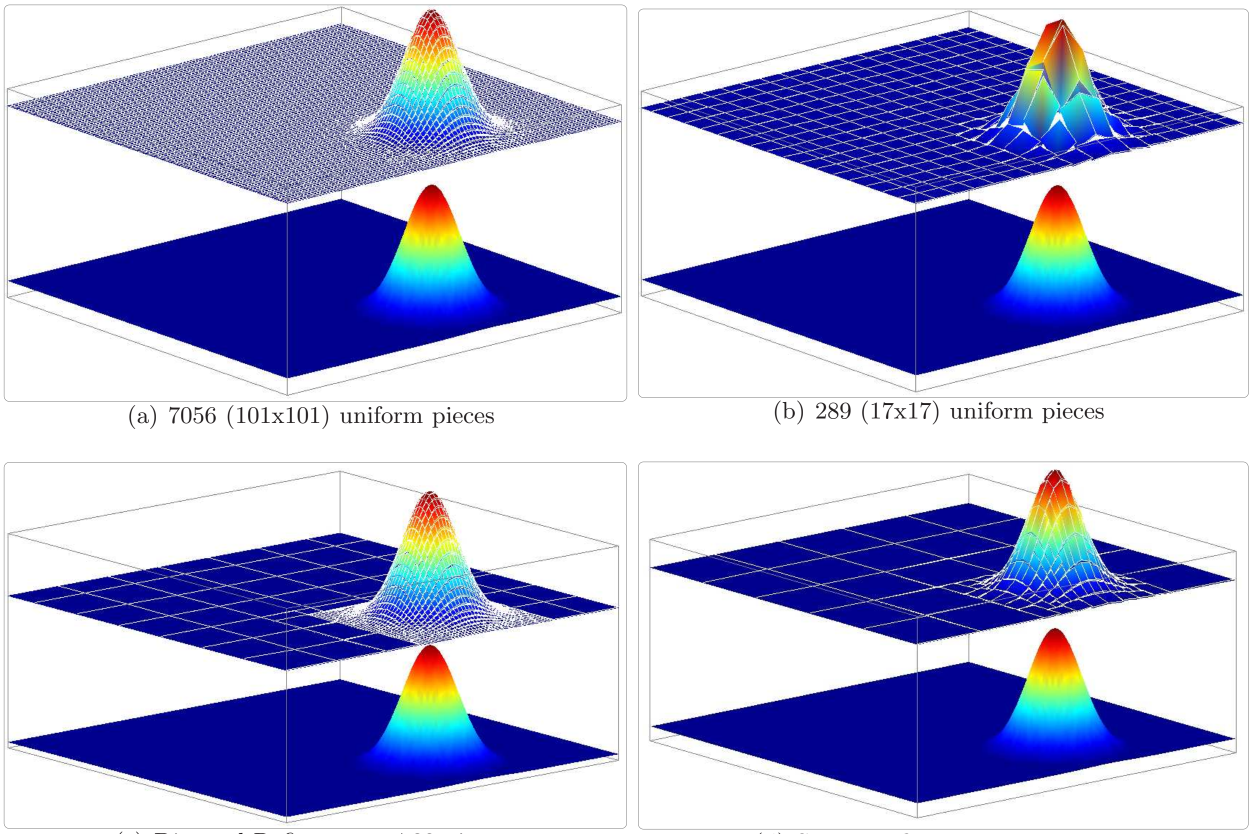

(a) 7056 (101x101) uniform pieces
(b) 289 (17x17) uniform pieces
(c) Directed Refinement - 732 pieces
(d) Smart Refinement - 225 pieces

Figure 17: **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.

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 $f_i(\overrightarrow{x})$, and the weight of each function is given by w_i, the solution to a problem with k functions is given by:
The algorithm is described in detail in [3], 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

$$\vec{x}^* = \underset{\vec{x}}{\operatorname{argmax}} \sum_{i=0}^{k-1} w_i f_i(\vec{x})$$

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 fully expanded tree, and utilizes proper pruning to guarantee global optimality. The algorithm does allow for a parameter for guaranteed limited back-off from 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 described in Section 8. The output shown below in Listing 11 is an excerpt of the full output shown in Listing 14 on page 83. 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 11 - Example `uHelmScope` *output containing information about the IvP solver.*

```
1  ==============   uHelmScope Report  ============== ENGAGED  (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)
```

```
  Halted:         false   (0 warnings)
Helm Decision: [speed,0,4,21] [course,0,359,360]
  speed = 3.00
  course = 177.00
Behaviors Active: ---------- (1)
  waypt_survey (13.0) (pwt=100.00) (pcs=1227) (cpu=0.01) (upd=0/0)
Behaviors Running: --------- (0)
Behaviors Idle: ------------ (1)
  waypt_return (22.8)
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 Standard and Overloadable 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 Chapter 6. The focus in this section is an expansion of detail of Step 3 in Figure 11 on page 52.

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 18. 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.

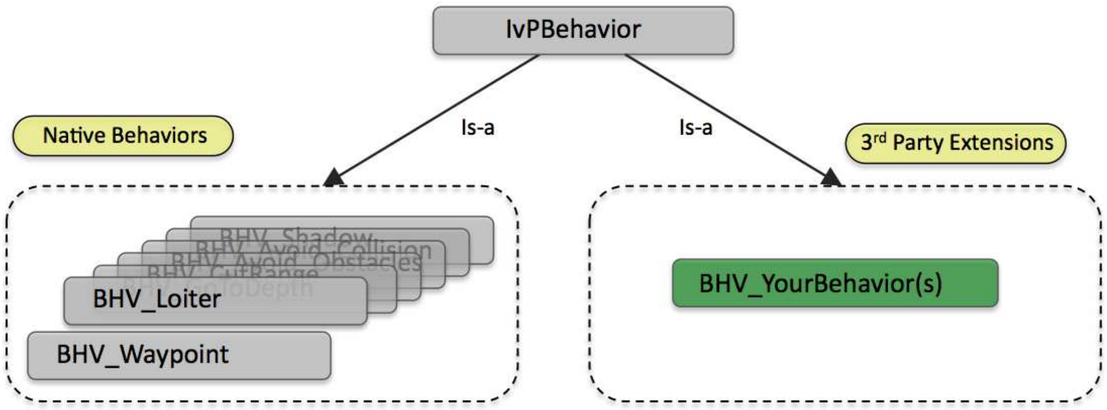


Figure 18: **Behavior inheritance**: Behaviors are derived from the `IvPBehavior` superclass. The native behaviors are the behaviors distributed with the helm. New behaviors also need to be subclass of the IvPBehavior class to work with the helm. Certain virtual functions invoked by the helm may be optionally 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 three 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.

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.

`NAME:` The name of the behavior - should be unique between all behaviors. Duplicates may be confusing, but should not cause helm errors. 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 7.2.3 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 7.2.3 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 7.2.3 for more detail.

`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 7.2.3 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 `((`$K \leq 4$`)`. 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 a equal-separated pair such as `TRANSITING`=true. More then one flag may be provided. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 62 for more detail on posting flags to the MOOSDB from the helm.

`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 an equal-separated pair such as `WAITING`=true. More then one flag may be provided. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 62 for more detail on posting flags to the MOOSDB from the helm.

`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 an equal-separated pair such as `TRANSITING`=true. More then one flag may be provided. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 62 for more detail on posting flags to the MOOSDB from the helm.

`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 a equal-separated pair such as `OUT_OF_RANGE`=true. More then one flag may be provided. These can be used to satisfy or block the conditions of other behaviors. See Section 6.5.4 on page 62 for more detail on posting flags to the MOOSDB from the helm.

`ENDFLAG`: 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. See Section 6.5.4 on page 62 for more detail on posting flags to the MOOSDB from the helm.

`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 runtime. 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 7.2.2 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 7.2.5 on page 75 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 7.2.4 for more detail.

7.2.2 Altering Behavior 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 12. 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 12 - An example behavior configuration using the `UPDATES` *parameter.*

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

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 7 in Listing 12). 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 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.

7.2.3 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. If left unspecified, there is no time limit to the behavior. By default, the duration clock begins ticking as soon as the helm engages. The duration clock remains ticking when or if the behavior subsequently enters the idle state. It even remains ticking if the helm temporarily disengages. 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 it's condtions 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.

7.2.4 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.

7.2.5 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 `"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. An implementation of the `setParam()` function for a new behavior should attempt to first handle a parameter-value pair at the `IvPBehavior` level, and only handle it locally if it is not recognized at the superclass level. The example below in Listing 13 gives an example for a fictional behavior `BHV_YourBehavior` having a single parameter `period`.

Listing 13 - An example `setParam()` *implementation for fictional* `BHV_YourBehavior`.

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

Since the `period` parameter refers to a time period, a check is made on line 4 that the value component indeed is a positive 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.) 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 19. 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. An *idle* behavior is one that has not mets 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 19:

- 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 19 regarding the sequence of these function calls.

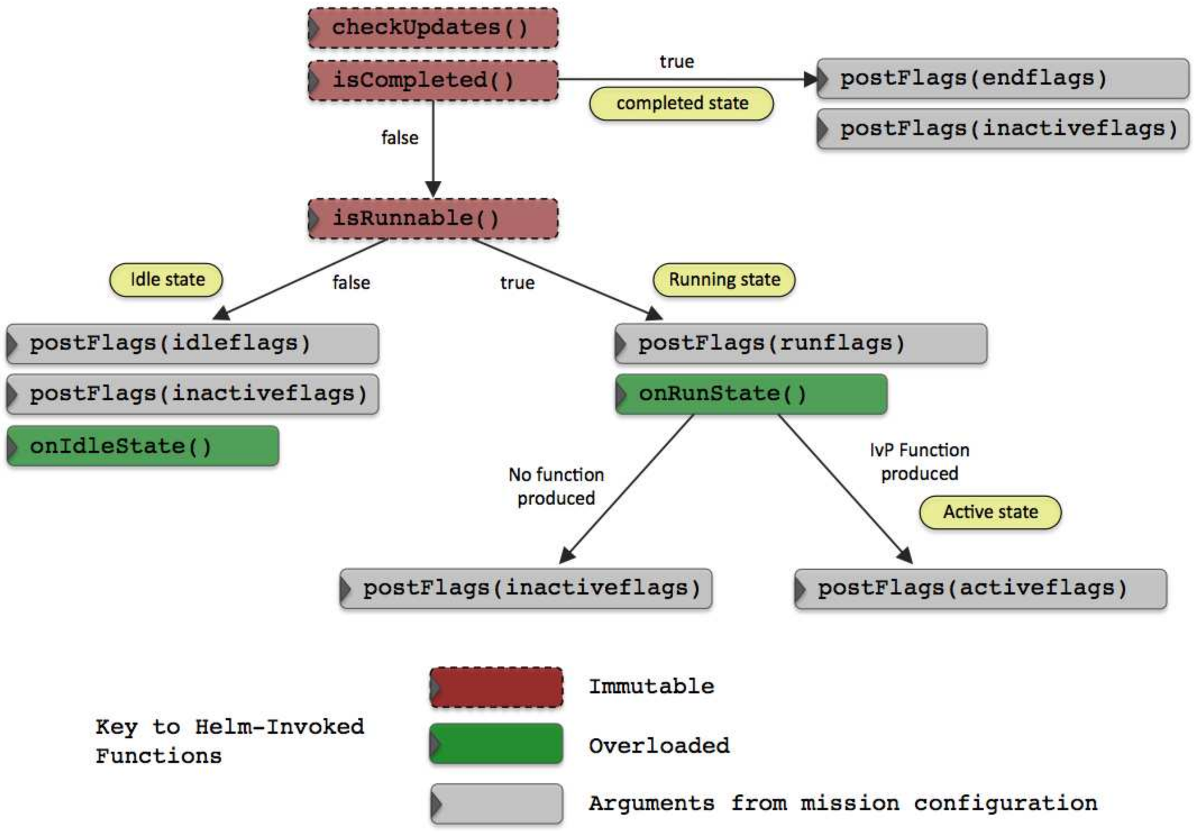


Figure 19: **Behavior function-calls by the helm**: The helm invokes a sequence of functions on each behavior on each iteration of the helm. The sequence of calls is dependent on what the behavior returns, and reflects the behaviors activity state. Certain functions are immutable and can not be overloaded 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.

`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 7.2.2 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 7.2.5). (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()`: The `onRunState()` function is called by the helm when deemed to be in the *running* state (Figure 19). 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 19). 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 implemenation 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`.

`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 issue 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 5.6 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 5.6 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 5.6 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 5.6 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.

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:

```
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.

```
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 timestamp indicating the last time the variable was changed in the buffer can be obtained from the following function call:

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

The string argument is the variable name, and the return value is cumulative time in seconds since the helm was launched. 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:

```
string  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 `Iterate()` loop prior to processing all new updates to the buffer from the MOOS mail stack. Thus the timestamp returned by the above call should be exactly the same for successive calls by all behaviors within a helm iteration, and the timestamps returned by `getBufferTimeVal()` and `getBufferCurrTime()` should be exactly the same if the variable was updated by new mail received by the helm at the beginning of the current 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:

```
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:

```
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 19 on page 77. 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 in a separate document.

8 uHelmScope

8.1 Brief Overview

The uHelmScope application is a console based tool for monitoring output of the IvP helm, i.e., the pHelmIvP process. The helm produces a few key MOOS variables on each iteration that pack in a substantial amount of information about what happened during a particular iteration. The helm scope subscribes for and parses this information, and writes it to standard output in a console window for the user to monitor. The user can dynamically pause or alter the output format to suit one's needs, and multiple scopes can be run simultaneously. The helm scope in no way influences the performance of the helm - it is strictly a passive observer.

8.2 Console Output of uHelmScope

The example console output shown in Listing 14 is used for explaining the uHelmScope fields.

Listing 14 - Example uHelmScope *output.*

```
 1  ==============   uHelmScope Report  ============== ENGAGED  (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) (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)
18
19  #  MOOSDB-SCOPE ------------------------------------ (Hit '#' to en/disable)
20  #
21  #  VarName           Source       Time     Community  VarValue
22  #  ----------------  -----------  -------  ---------  -----------
23  #  BHV_WARNING        n/a          n/a      n/a       n/a
24  #  AIS_REPORT_LOCAL  pTrans..rAIS 24.32    alpha      "NAME=alpha,TYPE=KAYAK,MOOSDB"+
25  #  DEPLOY*           iRemote      11.25    alpha      "true"
26  #  RETURN*           pHelmIvP     5.21     alpha      "false"
27
28  @  BEHAVIOR-POSTS TO MOOSDB ----------------------- (Hit '@' to en/disable)
29  @
30  @  MOOS Variable     Value
31  @  -------------     -------  (BEHAVIOR=waypt_survey)
32  @  PC_waypt_survey   -- ok --
33  @  WPT_STAT_LOCAL    vname=alpha,index=1,dist=80.47698,eta=26.83870
34  @  WPT_INDEX         1
35  @  VIEW_SEGLIST      label,alpha_waypt_survey : 30,-20:30,-100:90,-100: +
36  @  -------------     -------  (BEHAVIOR=waypt_return)
37  @  PC_waypt_return   RETURN = true
38  @  VIEW_SEGLIST      label,alpha_waypt_return : 0,0
39  @  VIEW_POINT        0,0,0,waypt_return
```

There are three groups of information in the uHelmScope output on each report to the console - the general helm overview (lines 1-17), a MOOSDB scope for a select subset of MOOS variables (lines

19-26), and a report on the MOOS variables published by the helm on the current iteration (lines 28-39). The output of each group is explained in the next three subsections.

8.2.1 The General Helm Overview Section of the `uHelmScope` Output

The first block of output produced by `uHelmScope` provides an overview of the helm. This is lines 1-17 in Listing 14, but the number of lines may vary with the mission and state of mission execution. The integer value at the end of line 1 indicates the number of `uHelmScope` reports written to the console. This can confirm to the user that an action that should result in a new report generation has indeed worked properly. The integer on line 2 is the counter kept by the helm, incremented on each helm iteration. The three sets of numbers that follow indicate the observed time between helm iterations. These numbers are reported by the helm and are not inferred by the scope. The first number is the average over the most recent five iterations. The second is the average over the most recent 58 iterations. The last is the maximum helm-reported interval observed by the scope. The number of iterations used to generate the first two numbers can be set by the user in the `uHelmScope` configuration block. The default is 5 and 100 respectively. The number 58 is shown in the second group simply because 100 iterations hadn't been observed yet. The helm is apparently only on iteration 66 in this example and `uHelmScope` apparently didn't start and connect to the `MOOSDB` until the helm was on iteration 8.

The value on Line 3 represents the the number of IvP functions produced by the active helm behaviors, one per active behavior. The solve-time on line 5 represents the time, in seconds, needed to solve the IvP problem comprised the *n* IvP functions. The number that follows in parentheses is the maximum solve-time observed by the scope. The create-time on line 6 is the total time needed by all active behaviors to produce their IvP function output. The loop time on line 7 is simply the sum of lines 5 and 6. The Boolean on line 8 is true only if the helm is halted on an emergency or critical error condition. Also on line 8 is the number of warnings generated by the helm. This number is reported by the helm and *not* simply the number of warnings observed by the scope. This number coincides with the number of times the helm writes a new message to the variable `BHV_WARNING`.

The helm decision space (i.e., IvP domain) is displayed on line 9, with the following lines used to display the actual helm decision. Following this is a list of all the active, running, idle and completed behaviors. At any point in time, each instantiated IvP behavior is in one of these four states and each behavior specified in the behavior file should appear in one of these groups. Technically all *active* behaviors are also *running* behaviors but not vice versa. So only the running behaviors that are not active (i.e., the behaviors that could have, but chose not to produce an objective function), are listed in the "`Behaviors Running:`" group. Immediately following each behavior the time, in seconds, that the behavior has been in the current state is shown in parentheses. For the active behaviors (see line 13) this information is followed by the priority weight of the behavior, the number of pieces in the produced IvP function, and the amount of CPU time required to build the function. If the behavior also is accepting dynamic parameter updates the last piece of information on line 13 shows how many successful updates where made against how many attempts. A failed update attempt also generates a helm warning, counted on line 8. The idle and completed behaviors are listed by default one per line. This can be changed to list them on one long line by hitting the 'b' key interactively. Insight into why an idle behavior is not in the running state can be found in the another part of the report (e.g., line 37) described below in Section 8.2.3.

8.2.2 The MOOSDB-Scope Section of the `uHelmScope` Output

Part of understanding what is happening in the helm involves the monitoring of variables in the `MOOSDB` that can either affect the helm or reveal what is being produced by the helm. Although there are other MOOS scope tools available (e.g., uXMS or uMS), this feature does two things the other scopes do not. First, it is simply a convenience for the user to monitor a few key variables in the same screen space. Second, `uHelmScope` automatically registers for the variables that the helm reasons over to determine the behavior activity states. It will register for all variables appearing in behavior conditions, runflags, activeflags, inactiveflags, endflags and idleflags. Variables that are registered for by this criteria are indicated by an asterisk at the end of the variable name. If the output resulting from these automatic registrations becomes unwanted, it can be toggled off by typing 's'.

The lines comprising the MOOSDB-Scope section of the `uHelmScope` output are all preceded by the '#' character. This is to help discern this block from the others, and as a reminder that the whole block can be toggled off and on by typing the '#' character. The columns in Listing 14 are truncated to a set maximum width for readability. The default is to have truncation turned off. The mode can be toggled by the console user with the 't' character, or set in the MOOS configuration block or with a command line switch. A truncated entry in the `VarValue` column has a '+' at the end of the line. Truncated entries in other columns will have ".." embedded in the entry. Line 24 shows an example of both kinds of truncation.

The variables included in the scope list can be specified in the `uHelmScope` configuration block of a MOOS file. In the MOOS file, the lines have the form:

```
VAR = VARIABLE_1, VARIABLE_2, VARIABLE_3, ...
```

An example configuration is given in Listing 17. Variables can also be given on the command line. Duplicates requests, should they occur, are simply ignored. Occasionally a console user may want to suppress the scoping of variables listed in the MOOS file and instead only scope on a couple variables given on the command line. The command line switch `-c` will suppress the variables listed in the MOOS file - unless a variable is also given on the command line. In line 23 of Listing 14, the variable `BHV_WARNING` is a *virgin* variable, i.e., it has yet to be written to by any MOOS process and shows `n/a` in the four output columns. By default, virgin variables are displayed, but their display can be toggled by the console user by typing '-v'.

8.2.3 The Behavior-Posts Section of the `uHelmScope` Output

The Behavior-Posts section is the third group of output in `uHelmScope` lists MOOS variables and values posted by the helm on the current iteration. Each variable was posted by a particular helm behavior and the grouping in the output is accordingly by behavior. Unlike the variables in the MOOSDB-Scope section, entries in this section only appear if they were written to on the current iteration. The lines comprising the Behavior-Posts section of the `uHelmScope` output are all preceded by the '@' character. This is to help discern this block from the others, and as a reminder that the whole block can be toggled off and on by typing the '@' character. As with the output in the MOOSDB-Scope output section, the output may be truncated. A trailing '+' at the end of the line indicates the variable value has been truncated.

There are a few switches for keeping the output in this section concise. A behavior posts a few standard MOOS variables on every iteration that may be essentially clutter for users in most cases. A behavior FOO for example produces the variables PWT_FOO, STATE_FOO, and UH_FOO which indicate the priority weight, run-state, and tally of successful updates respectively. Since this information is present in other parts of the uHelmScope output, these variables are by default suppressed in the Behavior-Posts output. Two other standard variables are PC_FOO and VIEW_* which indicate the precondition keeping a behavior in an idle state, and standard viewing hints to a rendering engine. Since this information is not present elsewhere in the uHelmScope output, it is not masked out by default. A console user can mask out the PWT, STATE_* and UH_* variables by typing 'm'. The PC_* and VIEW_* variables can be masked out by typing 'M'. All masked variables can be unmasked by typing 'u'.

8.3 Stepping Forward and Backward Through Saved Scope History

The user has the option of pausing and stepping forward or backward through helm iterations to analyse how a set of events may have unfolded. Stepping one event forward or backward can be done with the '[' and ']' keys respectively. Stepping 10 or 100 events can be done with the '{' and '}, and '(' and ')' keys respectively. The current helm iteration being displayed is always shown on the second line of the output. For each helm iteration, the uHelmScope process stores the information published by the helm (Section 8.5), and thus the memory usage of uHelmScope would grow unbounded if left unchecked. Therefore information is kept for a maximum of 2000 helm iterations. This number is *not* a configuration parameter - to preclude a user from inadvertently setting this too high and inducing the system maladies of a single process with runaway memory usage. To change this number, a user must change the source code (in particular the variable m_history_size_max in the file HelmScope.cpp). The uHelmScope history is therefore a moving window of fixed size that continues to shift right as new helm information is received. Stepping forward or backwards therefore is subject to the constraints of this window. Any steps backward or forward will in effect generate a new *requested* helm index for viewing. The requested index, if older than the oldest stored index, will be set exactly to the oldest stored index. Similarly in the other direction. It's quite possible then to hit the '[' key to step left by one index, and have the result be a report that is not one index older, but rather some number of indexes newer. Hitting the space bar or 'r' key always generates a report for the very latest helm information, with the 'r' putting the scope into streaming, i.e., continuous update, mode.

8.4 Console Key Mapping and Command Line Usage Summaries

The uHelmScope has a separate thread to accept user input from the console to adjust the content and format of the console output. It operates in either the *streaming mode*, where new helm summaries are displayed as soon as they are received, or the *paused mode* where no further output is generated until the user requests it. The key mappings can be summarized in the console output by typing the 'h' key, which also sets the mode to *paused*. The key mappings shown to the user are shown in Listing 15.

Listing 15 - Key mapping summary shown after hitting 'h' in a console.

```
KeyStroke  Function
---------  ---------------------------
```

```
     Spc     Pause and Update latest information once - now
     r/R     Resume information refresh
     h/H     Show this Help msg - 'r' to resume
     b/B     Toggle Show Idle/Completed Behavior Details
     t/T     Toggle truncation of column output
     m/M     Toggle display of Hiearchical Mode Declarations
      f      Filter PWT_* UH_* STATE_* in Behavior-Posts Report
      F      Filter PC_* VIEW_* in Behavior-Posts Report
     s/S     Toggle Behavior State Vars in MOOSDB-Scope Report
     u/U     Unmask all variables in Behavior-Posts Report
     v/V     Toggle display of virgins in MOOSDB-Scope output
     [/]     Display Iteration 1 step prev/forward
     {/}     Display Iteration 10 steps prev/forward
     (/)     Display Iteration 100 steps prev/forward
      #      Toggle Show the MOOSDB-Scope Report
      @      Toggle Show the Behavior-Posts Report

 Hit 'r' to resume outputs, or SPACEBAR for a single update
```

Several of the same preferences for adjusting the content and format of the `uHelmScope` output can be expressed on the command line, with a command line switch. The switches available are shown to the user by typing `uHelmScope -h`. The output shown to the user is shown in Listing 16.

Listing 16 - Command line usage of the `uHelmScope` *application.*

```
> uHelmScope -h
Usage: uHelmScope moosfile.moos [switches] [MOOSVARS]
  -t:  Column truncation is on (off by default)
  -c:  Exclude MOOS Vars in MOOS file from MOOSDB-Scope
  -x:  Suppress MOOSDB-Scope output block
  -p:  Suppress Behavior-Posts output block
  -v:  Suppress display of virgins in MOOSDB-Scope block
  -r:  Streaming (unpaused) output of helm iterations
  MOOSVAR_1 MOOSVAR_2 .... MOOSVAR_N
```

The command line invocation also accepts any number of MOOS variables to be included in the MOOSDB-Scope portion of the `uHelmScope` output. Any argument on the command line that does not end in `.moos`, and is not one of the switches listed above, is interpreted to be a requested MOOS variable for inclusion in the scope list. Thus the order of the switches and MOOS variables do not matter. These variables are added to the list of variables that may have been specified in the `uHelmScope` configuration block of the MOOS file. Scoping on *only* the variables given on the command line can be accomplished using the `-c` switch. To support the simultaneous running of more than one `uHelmScope` connected to the same `MOOSDB`, `uHelmScope` generates a random number N between 0 and 10,000 and registers with the `MOOSDB` as `uHelmScope_N`.

8.5 IvPHelm MOOS Variable Output Supporting `uHelmScope` Reports

There are six variables published by the `pHelmIvP` MOOS process, and registered for by the `uHelmScope` process, that provide critical information for generating `uHelmScope` reports. They are: `IVPHELM_SUMMARY`, `IVPHELM_POSTINGS`, `IVPHELM_ENGAGED`, `IVPHELM_STATEVARS`, `IVPHELM_DOMAIN`, and `IVPHELM_MODESET`. The first three are produced on each iteration of the helm, and the last three are typically only produced once when the helm is launched.

```
IVPHELM_SUMMARY  = "iter=66,ofnum=1,warnings=0,utc_time=1209755370.74,solve_time=0.00,
     create_time=0.02,loop_time=0.02,var=speed:3.0,var=course:108.0,halted=false,
     running_bhvs=none,active_bhvs=waypt_survey$6.8$100.00$1236$0.01$0/0,
     modes=MODE@ACTIVE:SURVEYING,idle_bhvs=waypt_return$55.3$n/a,completed_bhvs=none"

IVPHELM_POSTINGS  = "waypt_return$@!$66$@!$PC_waypt_return=RETURN = true$@!$VIEW_SEGLIST=label,
     alpha_waypt_return : 0,0$@!$VIEW_POINT=0,0,0,waypt_return$@!$PWT_BHV_WAYPT_RETURN=0
     $@!$STATE_BHV_WAYPT_RETURN=0"

IVPHELM_POSTINGS  = waypt_survey$@!$66$@!$PC_waypt_survey=-- ok --$@!$WPT_STAT_LOCAL=vname=alpha,
     index=1,dist=80.47698,eta=26.83870$@!$WPT_INDEX=1$@!$VIEW_SEGLIST=label,
     alpha_waypt_survey:30,-20:30,-100:90,-100:110,-60:90,-20$@!$PWT_BHV_WAYPT_SURVEY=100$@!$
     STATE_BHV_WAYPT_SURVEY=2

IVPHELM_DOMAIN = "speed,0,4,21:course,0,359,360"

IVPHELM_STATEVARS = "RETURN,DEPLOY"

IVPHELM_MODESET   = "---,ACTIVE#---,INACTIVE#ACTIVE,SURVEYING#ACTIVE,RETURNING"

IVPHELM_ENGAGED   = "ENGAGED"
```

The `IVPHELM_SUMMARY` variable contains all the dynamic information included in the general helm overview (top) section of the uHelmScope output. It is a comma-separated list of `var=val` pairs. The `IVP_DOMAIN` variable also contributes to this section of output by providing the IvP domain used by the helm. The `IVPHELM_POSTINGS` variable includes a list of MOOS variables and values posted by the helm for a given behavior. The helm writes to this variable once per iteration *for each behavior*. The `IVPHELM_STATEVARS` variable affects the MOOSDB-Scope section of the `uHelmScope` output by identifying which MOOS variables are used by behaviors in conditions, runflags, endflags and idleflags.

8.6 Configuration Parameters for `uHelmScope`

Configuration for `uHelmScope` amounts to specifying a set of parameters affecting the terminal output format. An example configuration is shown in Listing 17, with all values set to the defaults. Launching `uHelmScope` with a MOOS file that does not contain a `uHelmScope` configuration block is perfectly reasonable.

Listing 17 - An example `uHelmScope` *configuration block.*

```
//------------------------------------------------------------
// uHelmScope configuration block

ProcessConfig = uHelmScope
{
  AppTick    = 1
  CommsTick  = 1

  PAUSED             = true     // All Parameters and Parameter-Values
  HZ_MEMORY          = 5, 100   // are __NOT__ Case Sensitive
  DISPLAY_MOOS_SCOPE = true
  DISPLAY_BHV_POSTS  = true
  DISPLAY_VIRGINS    = true
  DISPLAY_STATEVARS  = true
  TRUNCATED_OUTPUT   = false
  BEHAVIORS_CONCISE  = false

  VAR = BHV_WARNING, AIS_REPORT_LOCAL   // MOOS Variable names
}                                       // __ARE__ Case Sensitive
```

Each of the parameters, with the exception of `HZ_MEMORY` can also be set on the command line, or interactively at the console, with one of the switches or keyboard mappings listed in Section 8.4. A parameter setting in the MOOS configuration block will take precedence over a command line switch. The `HZ_MEMORY` parameter takes two integer values, the second of which must be larger than the first. This is the number of samples used to form the average time between helm intervals, displayed on line 2 of the `uHelmScope` output.

8.7 Publications and Subscriptions for `uHelmScope`

Variables published by the `uHelmScope` application

- `NONE`

Variables subscribed for by the `uHelmScope` application

- `<USER-DEFINED>`: Variables identified for scoping by the user in the `uHelmScope` will be subscribed for. See Section 8.2.2.
- `<HELM-DEFINED>`: As described in Section 8.2.2, the variables scoped by `uHelmScope` include any variables involved in the preconditions, runflags, idleflags, activeflags, inactiveflags, and endflags for any of the behaviors involved in the current helm configuration.
- `IVPHELM_SUMMARY`: See Section 8.5.
- `IVPHELM_POSTINGS`: See Section 8.5.
- `IVPHELM_STATEVARS`: See Section 8.5.
- `IVPHELM_IVP_DOMAIN`: See Section 8.5.
- `IVPHELM_IVP_MODESET`: See Section 8.5.
- `IVPHELM_IVP_ENGAGED`: See Section 8.5.

9 Geometry Utilities

9.1 Brief Overview

This section discusses a few geometry data structures often used by the helm and the pMarineViewer application - convex polygons, lists of line segments, and points. These data structures are implemented by the classes `XYPolygon`, `XYSegList`, `XYPoint` respectively in the `lib_geometry` module distributed with the MOOS-IvP software bundle. The implementation of these class definitions is somewhat shielded from the helm user's perspective, but they are often involved in parameter settings of for behaviors. So the issue of how to specify a given geometric structure with a formatted string is discussed here.

Furthermore, the pMarineViewer application accepts these data structures for rendering by subscribing to three MOOS variables `VIEW_POLYGON`, `VIEW_SEGLIST`, and `VIEW_POINT`. These variables contain a string format representation of the structure, often with further visual hints on the color or size of the edges and vertices for rendering. These variables may originate from any MOOS application, but are also often posted by helm behaviors to provide visual clues about what is going on in the vehicle. In the Alpha mission described in Section 4 for example, the waypoint behavior posted a seglist representing the set of waypoints for which it was configured, as well as posting a point indicating the next point on the behavior's list to traverse.

9.2 Points

Points are implemented in the `XYPoint` class, and minimally represent a point in the x-y plane. These objects are used internally for applications and behaviors, and may also be involved in rendering in a GUI and therefore may have additional fields to support this.

9.2.1 String Representations for Points

The only required information for a point specification is its position in the x-y plane. A third value may optionally be specified in the z-plane. All four of the below string representations correspond to the very same data structure:

```
point   = 60,-40
point   = 60, -40, 0
point   = x=60, y=-40
point   = x=60, y=-40, z=0
```

9.2.2 Optional Point Parameters

Points also may have several optional fields associated with them. The `label` field is string that is often rendered with a point in MOOS GUI applications such as the pMarineViewer. The `label_color` field represents a color preference for the label rendering. The `type` and `source` fields are additional string fields for further distinguishing a point in applications that handle them. The `active` field is a Boolean that is used in the pMarineViewer application to indicate whether the the point should be rendered. The `time` field is a double that may optionally be set to indicate when the point was generated, or how long it should exist before "expiring", or however an application may wish to interpret it. The `vertex_color` and `vertex_size` field represent further rendering preferences. The following are two equivalent further string representations:

```
point   = x=60, y=-40, label=home, label_color=red, source=henry, type=waypoint,
          time=30, active=true, vertex_color=white, vertex_size=5
point   = 60,-40:label,home:label_color,red:source,henry:type,waypoint
          time,30:active,true:vertex_color,white:vertex_size,5
```

The former is a more user-friendly format for specifying a point, perhaps found in a configuration file for example. The latter is the string representation passed around internally when `XYPoint` objects are automatically converted to strings and back again in the code. This format is more likely to be found in log files or seen when scoping on variables with one of the MOOS scoping tools.

9.3 Polygons

Polygons are implemented in the `XYPolygon` class. This implementation accepts as a valid construction only specifications that build a convex polygon. Common operations used internally by behaviors and other applications, such as intersection tests, distance calculations etc, are greatly simplified and more efficient when dealing with convex polygons.

9.3.1 String Representations for Polygons

Polygons are defined by a set of vertices and the simplest way to specify the points is with a line comprised of a sequence of colon-separated pairs of comma-separated x-y points in local coordinates such as:

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

If one of the pairs, such as the last one above, contains the keyword `label` on the left, then the value on the right, e.g., `foxtrot` as above, is the label associated with the polygon. An alternative notation for the same polygon is given by the following:

```
polygon   = label=foxtrot, pts="60,-40:60,-160:150,-160:180,-100:150,-40"
```

This is an comma-separated list of equals-separated pairs. The ordering of the comma-separated components is insignificant. The points describing the polygon are provided in quotes to signify to the parser that everything in quotes is the right-hand side of the `pts=` component. Both formats are acceptable specifications of a polygon in a behavior for which there is a `polygon` parameter.

9.3.2 A Polygon String Representation using the Radial Format

Polygons may also be specified by their shape and the shape parameters. For example, a commonly used polygon is formed by points of an equal radial distance around a center point. The following is an example:

```
polygon   = format=radial, label=foxtrot, x=0, y=40, radius=60, pts=6, snap=1
```

The `snap` component in the above example signifies that the vertices should be rounded to the nearest 1-meter value. The `x`, `y` parameters specify the middle of the polygon, and `radius` parameters specify the distance from the center for each vertex. The `pts` parameters specifies the number of vertices used, as shown in Figure 20.

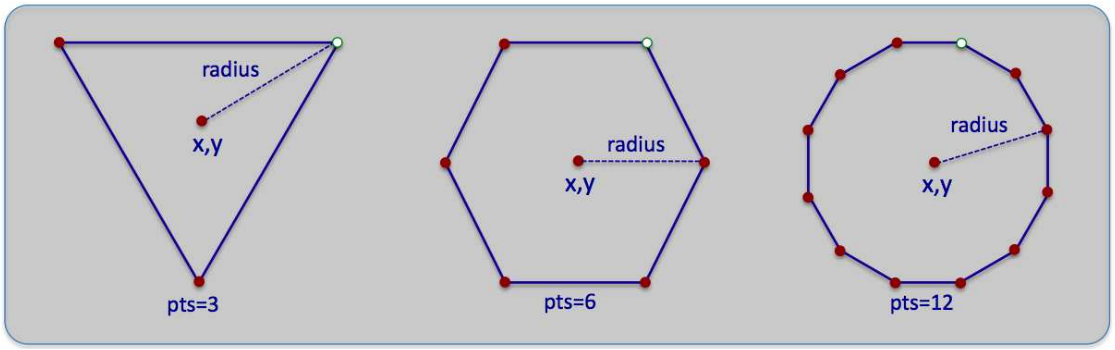


Figure 20: **Polygons built with the `radial` format:** Radial polygons are specified by (a) the location of their center, (b) the number of vertices, and (c) the radial distance from the center to each vertex. The lighter vertex in each polygon indicates the first vertex if traversing in sequence, proceeding clockwise.

9.3.3 A Polygon String Representation using the Ellipse Format

Polygons may also be built using the `ellipse` format. The following is an example:

```
polygon   = label=golf, format=ellipse, x=0, y=40, degs=45, pts=14, snap=1,
            major=100, minor=70
```

The `x`, `y` parameters specify the middle of the polygon, the `major` and `minor` parameters specify the radial distance of the major and minor axes. The `pts` parameters specifies the number of vertices used, as shown in Figure 21.

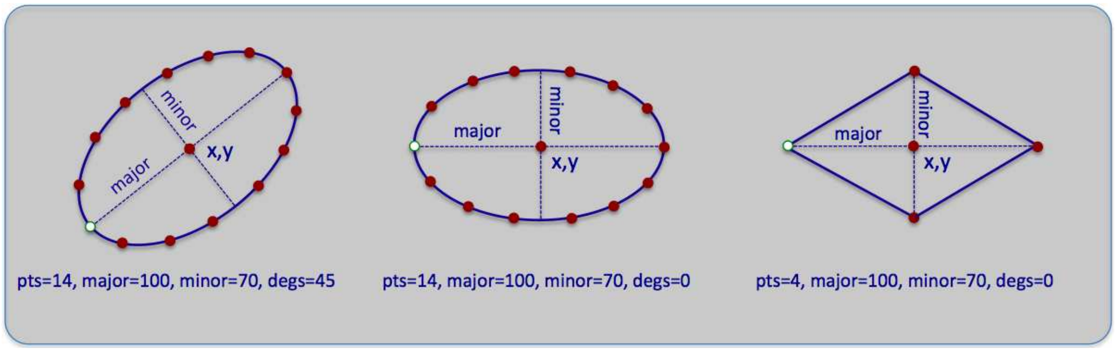

minor
major
x,y
pts=14, major=100, minor=70, degs=45
minor
major
x,y
pts=14, major=100, minor=70, degs=0
minor
major
x,y
pts=4, major=100, minor=70, degs=0

Figure 21: **Polygons built with the `ellipse` format:** Ellipse polygons are specified by (a) the location of their center, (b) the number of vertices, (c) the length of their major axis, (d) the length of their minor axis, and (e) the rotation of the ellipse.The lighter vertex in each polygon indicates the first vertex if traversing in sequence, proceeding clockwise.

The rotation of the ellipse can optionally be specified in radians. For example, `degs=45` is equivalent to `rads=0.785398`. If, for some reason, both are specified, the polygon will be built using the `rads` parameter. When using the ellipse format, a minimum `pts=4` must be specified.

9.3.4 Optional Polygon Parameters

Polygons also may have several optional fields associated with them. The `label` field is string that is often rendered with a polygon in MOOS GUI applications such as the pMarineViewer. The `label_color` field represents a color preference for the label rendering. The `type` and `source` fields are additional string fields for further distinguishing a polygon in applications that handle them. The `active` field is a Boolean that is used in the pMarineViewer application to indicate whether the the polygon should be rendered. The `time` field is a double that may optionally be set to indicate when the polygon was generated, or how long it should exist before "expiring", or however an application may wish to interpret it. The `vertex_color`, `edge_color`, and `vertex_size` fields represent further rendering preferences. The following are two equivalent further string representations:

```
polygon = format=radial, x=60, y=-40, radius=60, pts=8, snap=1, label=home,
          label_color=red, source=henry, type=survey, time=30, active=true,
          vertex_color=white, vertex_size=5, edge_size=2
polygon = format,radial:60,-40:radius,60:pts,8:snap,1:label,home:
          label_color,red:source,henry:type,survey:time,30:active,true:
          vertex_color,white:vertex_size,5:edge_size,2
```

The former is a more user-friendly format for specifying a polygon, perhaps found in a configuration file for example. The latter is the string representation passed around internally when `XYPolygon` objects are automatically converted to strings and back again in the code. This format is more likely to be found in log files or seen when scoping on variables with one of the MOOS scoping tools.

9.4 SegLists and their String Representations

9.4.1 Methods for Specifying Seglists

A *seglist* is a sequence of line segments given by a list of points. Seglists are specified with the `points` parameter, and the simplest way to specify the points is with a line comprised of a sequence of colon-separated pairs of comma-separated x-y points in local coordinates such as:

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

If one of the pairs, such as the last one above, contains the keyword `label` on the left, then the value on the right, e.g., `foxtrot` as above, is the label associated with the seglist.

9.4.2 A SegList String Representation using the Lawnmower Format

SegLists may also be built using the `lawnmower` format. The following is an example:

```
points   = format=lawnmower, label=foxtrot, x=0, y=40, height=60, width=180,
           lane_width=15, rows=north-south, degs=45
```

The rotation of the pattern can optionally be specified in radians. For example, `degs=45` is equivalent to `rads=0.785398`. If, for some reason, both are specified, the seglist will be built using the `rads` parameter.

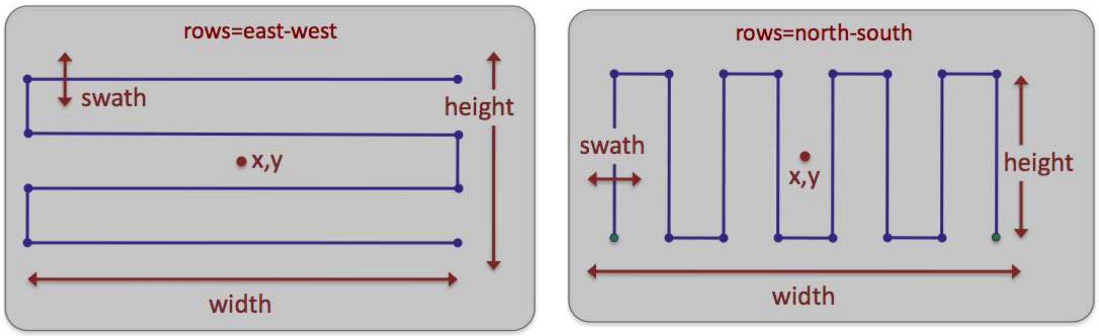


Figure 22: **SegLists built with the `lawnmower` format:** The pattern is specified by (a) the location of the center of the pattern, (b) the height and width of the pattern, (c) the lane width which determines the number of rows, (d) whether the pattern rows proceed north-south or east-west, and (e) an optional rotation of the pattern.

10 pMarineViewer

10.1 Brief Overview

The pMarineViewer application is a MOOS application written with FLTK and OpenGL for rendering vehicles and associated information and history during operation or simulation. The typical layout shown in Figure 23 is that pMarineViewer is running in its own dedicated local MOOS community while simulated or real vehicles on the water transmit information in the form of a stream of *node reports* to the local community.

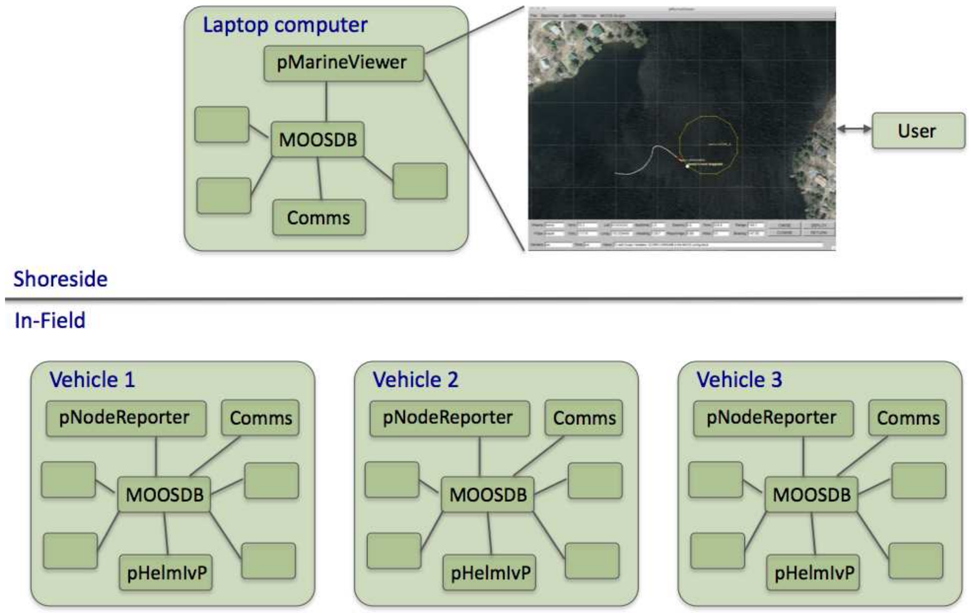


Figure 23: A common usage of the pMarineViewer is to have it running in a local MOOSDB community while receiving node reports on vehicle poise from other MOOS communities running on either real or simulated vehicles. The vehicles can also send messages with certain geometric information such as polygons and points that the view will accept and render.

The user is able manipulate a geo display to see multiple vehicle tracks and monitor key information about individual vehicles. In the primary interface mode the user is a passive observer, only able to manipulate what it sees and not able to initiate communications to the vehicles. However there are hooks available and described later in this section to allow the interface to accept field control commands.

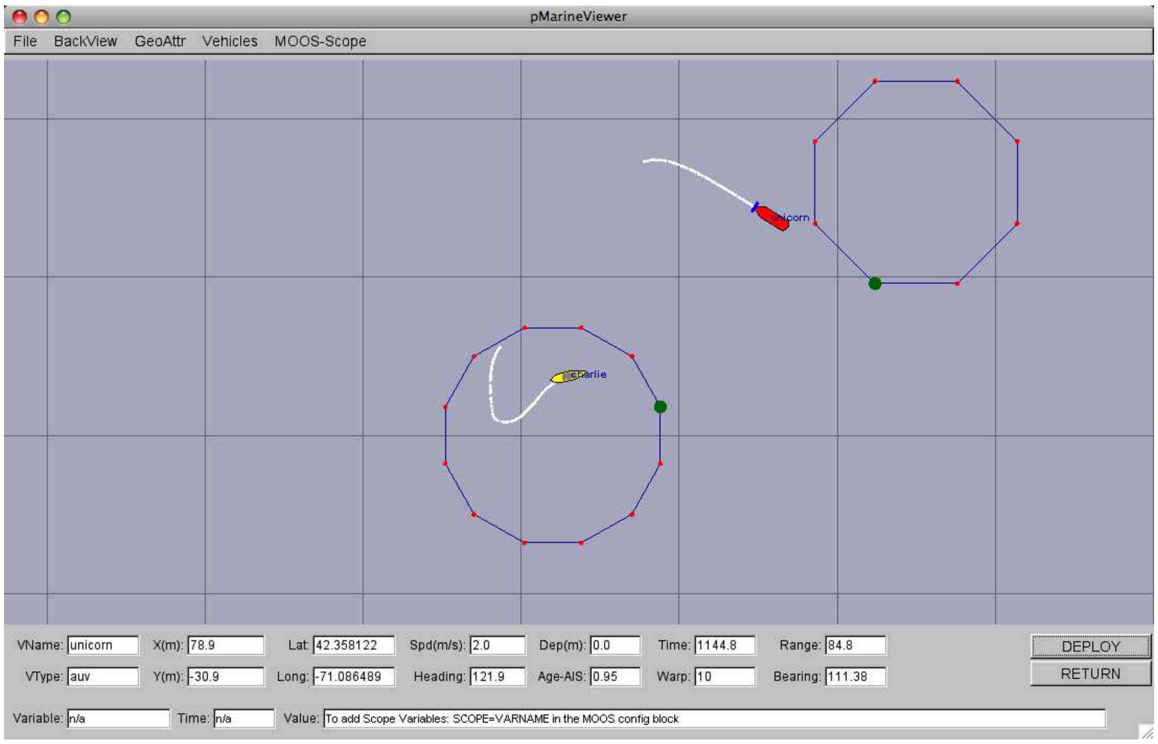
A key variable subscribed to by pMarineViewer is the variable NODE_REPORT, which has the following structure given by an example:

```
NODE_REPORT = "NAME=nyak201,TYPE=kayak,MOOSDB_TIME=53.049,UTC_TIME=1195844687.236,X=37.49,
              Y=-47.36, SPD=2.40,HDG=11.17,DEPTH=0"
```

Reports from different vehicles are sorted by their vehicle name and stored in histories locally in the pMarineViewer application. The NODE_REPORT is generated by the vehicles based on either sensor information, e.g., GPS or compass, or based on a local vehicle simulator.

10.2 Description of the pMarineViewer GUI Interface

The viewable area of the GUI has two parts - a geo display area where vehicles and perhaps other objects are rendered, and a lower area with certain data fields associated with an *active* vehicle are updated. A typical screen shot is shown in Figure 24 with two vehicles rendered - one AUV and one kayak. Vehicle labels and history are rendered. Properties of the vehicle rendering such as the trail length, size, and color, and vehicle size and color, and pan and zoom can be adjusted dynamically in the GUI. They can also be set in the `pMarineViewer` MOOS configuration block. Both methods of tuning the rendering parameters are described later in this section.



VName: unicorn X(m): 78.9 Lat: 42.358122 Spd(m/s): 2.0 Dep(m): 0.0 Time: 1144.8 Range: 84.8
VType: auv Y(m): -30.9 Long: -71.086489 Heading: 121.9 Age-AIS: 0.95 Warp: 10 Bearing: 111.38
Variable: n/a Time: n/a Value: To add Scope Variables: SCOPE=VARNAME in the MOOS config block

Figure 24: A screen shot of the `pMarineViewer` application running with two vehicles - one kayak platform, and one AUV platform. The charlie kayak platform is the *active* platform meaning the data fields on the bottom reflect the data for this platform.

The lower part of the display is dedicated to displaying detailed position information on a single *active* vehicle. Changing the designation of which vehicle is active can be accomplished by repeatedly hitting the 'v' key. The active vehicle is always rendered as red, while the non-active vehicles have a default color of yellow. Individual vehicle colors can be given different default values (even red, which could be confusing) by the user. The individual fields are described below:

- `VName:` The name of the active vehicle associated with the data in the other GUI data fields. The active vehicle is typically indicated also by changing to the color red on the geo display.
- `VType:` The platform type, e.g., AUV, Glider, Kayak, Ship or Unknown.
- `X(m):` The x (horizontal) position of the active vehicle given in meters in the local coordinate system.
- `Y(m):` The y (vertical) position of the active vehicle given in meters in the local coordinate system.

- `Lat:` The latitude (vertical) position of the active vehicle given in decimal latitude coordinates.
- `Lon:` The longitude (horizontal) position of the active vehicle given in decimal longitude coordinates.
- `Speed:` The speed of the active vehicle given in meters per second.
- `Heading:` The heading of the active vehicle given in degrees ($0 - 359.99$).
- `Depth:` The depth of the active vehicle given in meters.
- `Report-AGE:` The elapsed time in seconds since the last received node report for the active vehicle.
- `Time:` Time in seconds since the pMarineViewer process launched.
- `Warp:` The MOOS Time-Warp value. Simulations may run faster than real-time by this warp factor. MOOSTimeWarp is set as a global configuration parameter in the `.moos` file.
- `Range:` The range (in meters) of the active vehicle to a reference point. By default, this point is the datum, or the (0,0) point in local coordinates. The reference point may also be set to another particular vehicle. See Section 10.3.7 on the `ReferencePoint` pull-down menu.
- `Bearing:` The bearing (in degrees) of the active vehicle to a reference point. By default, this point is the datum, or the (0,0) point in local coordinates. The reference point may also be set to another particular vehicle. See Section 10.3.7 on the `ReferencePoint` pull-down menu.

In simulation, the age of the node report is likely to remain zero as shown in the figure, but when operating on the water, monitoring the node report age field can be the first indicator when a vehicle has failed or lost communications. Or it can act as an indicator of comms quality.

The lower three fields of the window are used for scoping on a single MOOS variable. See Section 10.3.4 for information on how to configure the `pMarineViewer` to scope on any number of MOOS variables and select a single variable via an optional pull-down menu. The scope fields are:

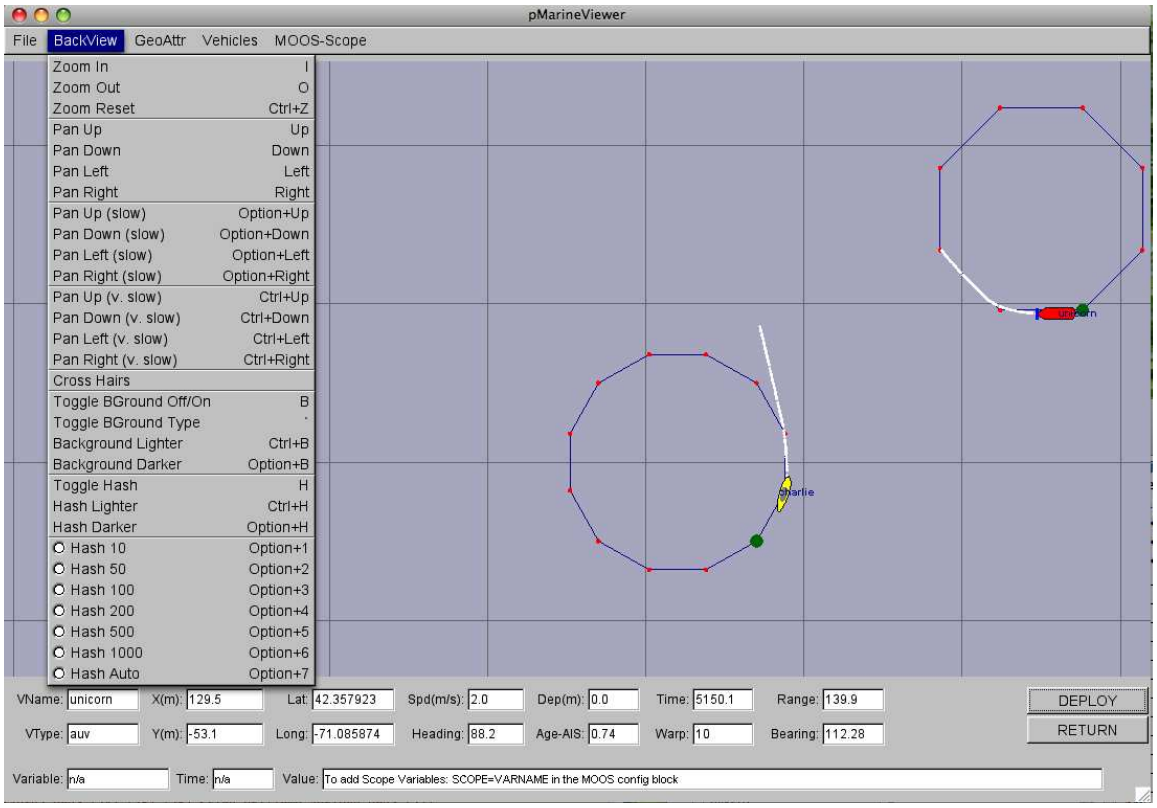
- `Variable:` The variable name of the MOOS variable currently being scoped, or `"n/a"` if no scope variables are configured.
- `Time:` The variable name of the MOOS variable currently being scoped, or `"n/a"` if no scope variables are configured.

10.3 Pull-Down Menu Options

Properties of the geo display rendering can be tuned to better suit a user or circumstance or for situations where screen shots are intended for use in other media such as papers or PowerPoint. There are two pull-down menus - the first deals with background properties, and the second deals with properties of the objects rendered on the foreground. Many of the adjustable properties can be adjusted by two other means besides the pull-down menus - by the hot keys defined for a particular pull-down menu item, or by configuring the parameter in the MOOS file configuration block.

10.3.1 The "BackView" Pull-Down Menu

Most pull-down menu items have hot keys defined (on the right in the menu). For certain actions like pan and zoom, in practice the typical user quickly adopts the hot-key interface. But the pull-down menu is one way to have a form of hot-key documentation always handy. The zooming commands affect the viewable area and apparent size of the objects. Zoom in with the 'i' or 'I' key, and zoom out with the 'o' or 'O' key. Return to the original zoom with ctrl+'z'.



VName: unicorn X(m): 129.5 Lat: 42.357923 Spd(m/s): 2.0 Dep(m): 0.0 Time: 5150.1 Range: 139.9

VType: auv Y(m): -53.1 Long: -71.085874 Heading: 88.2 Age-AIS: 0.74 Warp: 10 Bearing: 112.28

Variable: n/a Time: n/a Value: To add Scope Variables: SCOPE=VARNAME in the MOOS config block

Figure 25: **The BackView menu:** This pull-down menu lists the options, with hot-keys, for affecting rendering aspects of the geo-display background.

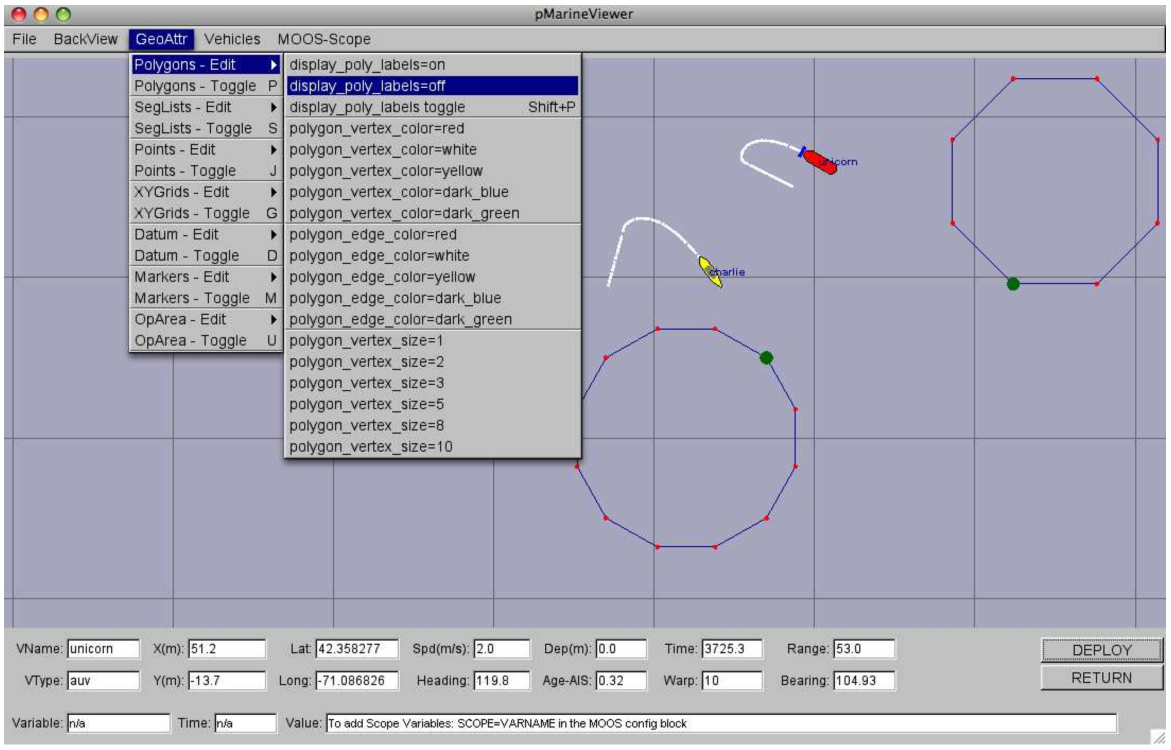
Panning is done with the keyboard arrow keys. Three rates of panning are supported. To pan in 20 meter increments, just use the arrow keys. To pan "slowly" in one meter increments, use the Alt + arrow keys. And to pan "very slowly", in increments of a tenth of a meter, use the Ctrl + arrow keys. The viewer supports two types of "convenience" panning. It will pan put the active vehicle in the center of the screen with the 'C' key, and will pan to put the average of all vehicle positions at the center of the screen with the 'c' key. These are part of the 'Vehicles' pull-down menu discussed in Section 10.3.3.

The background can be in one of two modes; either displaying a gray-scale background, or displaying a geo image read in as a texture into OpenGL from an image file. The default is the geo display mode if provided on start up, or the grey-scale mode if no image is provided. The mode can be toggled by typing the 'b' or 'B' key. The geo-display mode can have two sub-modes if two image files are provided on start-up. More on this in Section 10.7. This is useful if the user has access to a satellite image *and* a map image for the same operation area. The two can be toggled by hitting the back tick key. When in the grey-scale mode, the background can be made lighter by hitting the ctrl+'b' key, and darker by hitting the alt+'b' key.

Hash marks can be overlaid onto the background. By default this mode is off, but can be toggled with the 'h' or 'H' key. The hash marks are drawn in a grey-scale which can be made lighter by typing the ctrl+'h' key, and darker by typing the alt+'h' key. Certain hash parameters can also be set in the `pMarineViewer` configuration block of the MOOS file. The `hash_view` parameter can be set to either `true` or `false`. The default is `false`. The `hash_delta` parameter can be set to any integer in the range [10, 1000]. The default is 100.

10.3.2 The "GeoAttributes" Pull-Down Menu

The GeoAttributes pull-down menu allows the user to edit the properties of geometric objects capable of being rendered by the `pMarineViewer`. In general the Polygon, SegList, Point, and XYGrid objects are received by the viewer at run time to reflect artifacts generated by the IvP Helm indicating aspects of progress during their mission. The polygons in Figure 26 for example represents the set of waypoints being used by the vehicles shown.



VName: unicorn X(m): 51.2 Lat: 42.358277 Spd(m/s): 2.0 Dep(m): 0.0 Time: 3725.3 Range: 53.0

VType: auv Y(m): -13.7 Long: -71.086826 Heading: 119.8 Age-AIS: 0.32 Warp: 10 Bearing: 104.93

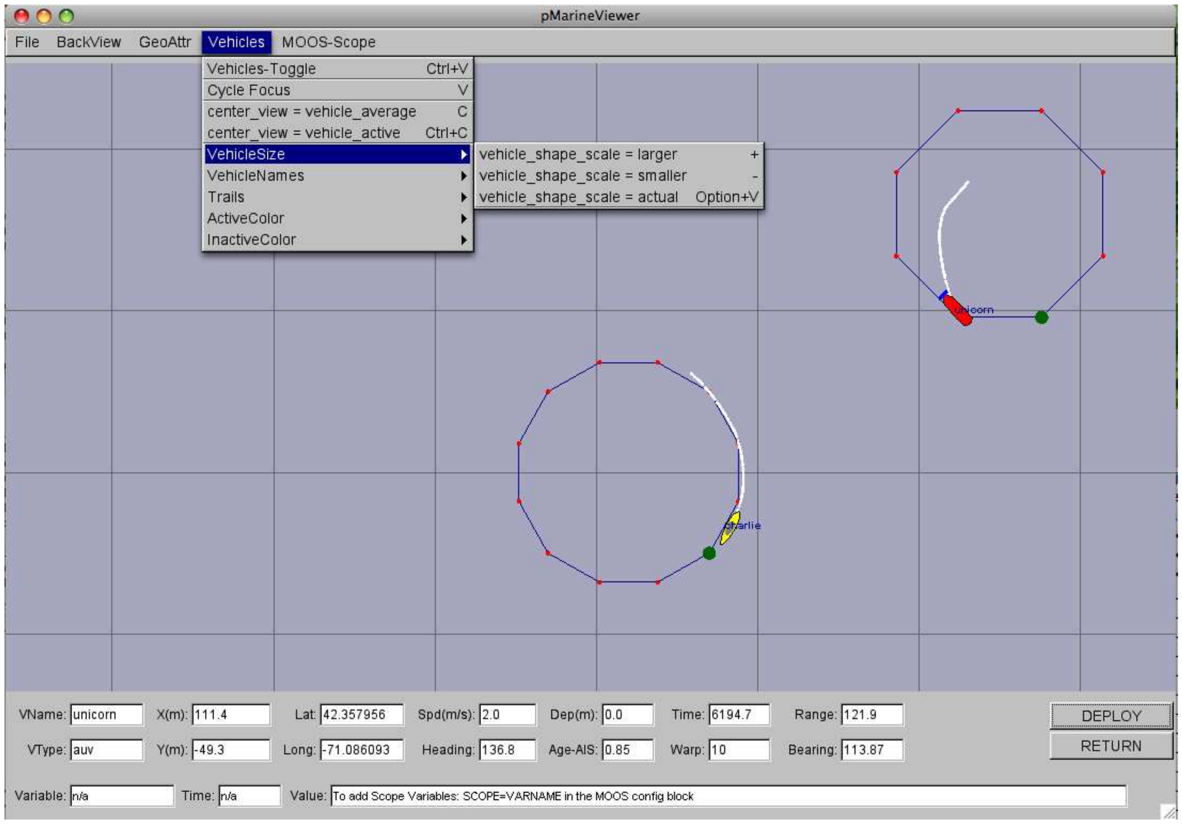
Variable: n/a Time: n/a Value: To add Scope Variables: SCOPE=VARNAME in the MOOS config block

Figure 26: **The GeoAttributes menu:** This pull-down menu lists the options and hot keys for affecting the rendering of geometric objects.

The Datum, Marker and OpArea objects are typically read in once at start-up and reflect persistent info about the operation area. The datum is a single point that represents (0,0) in local coordinates. Marker objects typically represent physical objects in the environment such as a buoy, or a fixed sensor. The OpArea objects are typically a combination of points and lines that reflect a region of earth where a set of vehicles are being operated. Each category has a hot key that toggles the rendering of all objects of the same type, and a secondary drop-down menu as shown in the figure that allows the adjustment of certain rendering properties of objects. Many of the items in the menu have form `parameter = value`, and these settings can also be achieved by including this line in the `pMarineViewer` configuration block in the MOOS file.

10.3.3 The "Vehicles" Pull-Down Menu

The *Vehicles* pull-down menu deals with properties of the objects displayed in the geo display foreground. The `Vehicles-Toggle` menu item will toggle the rendering of all vehicles and all trails. The `Cycle_Focus` menu item will set the index of the *active* vehicle, i.e., the vehicle who's attributes are being displayed in the lower output boxes. The assignment of an index to a vehicle depends on the arrival of node reports. If an node report arrives for a previously unknown vehicle, it is assigned a new index.


pMarineViewer
File
BackView
GeoAttr
Vehicles
MOOS-Scope
Vehicles-Toggle Ctrl+V
Cycle Focus V
center_view = vehicle_average C
center_view = vehicle_active Ctrl+C
VehicleSize
VehicleNames
Trails
ActiveColor
InactiveColor
vehicle_shape_scale = larger +
vehicle_shape_scale = smaller -
vehicle_shape_scale = actual Option+V
unicorn
charlie
DEPLOY
RETURN

VName: unicorn X(m): 111.4 Lat: 42.357956 Spd(m/s): 2.0 Dep(m): 0.0 Time: 6194.7 Range: 121.9

VType: auv Y(m): -49.3 Long: -71.086093 Heading: 136.8 Age-AIS: 0.85 Warp: 10 Bearing: 113.87

Variable: n/a Time: n/a Value: To add Scope Variables: SCOPE=VARNAME in the MOOS config block

Figure 27: **The ForeView menu:** this pull-down menu of the `pMarineViewer` lists the options, with hot-keys, for affecting rendering aspects of the objects on the geo-display foreground, such as vehicles and vehicle track history.

The `center_view` menu items alters the center of the view screen to be panned to either the position of the active vehicle, or the position representing the average of all vehicle positions. Once the user has selected this, this mode remains *sticky*, that is the viewer will automatically pan as new vehicle information arrives such that the view center remains with the active vehicle or the vehicle average position. As soon as the user pans manually (with the arrow keys), the viewer breaks from trying to update the view position in relation to received vehicle position information. The rendering of the vehicles can made larger with the '+' key, and smaller with the '-' key, as part of the `VehicleSize` pull-down menu as shown. The size change is applied to all vehicles equally as a scalar multiplier. Currently there is no capability to set the vehicle size individually, or to set the size automatically to scale.

Vehicle trail (track history) rendering can be toggled off and on with the 't' or 'T' key. The default is on. A set of predefined trail colors can be toggled through with the CTRL+'t' key. The individual trail points can be rendered with a line connecting each point, or by just showing the points. When the node report stream is flowing quickly, typically the user doesn't need or want to connect the points. When the viewer is accepting input from an AUV with perhaps a minute or longer delay in between reports, the connecting of points is helpful. This setting can be toggled with the 'y' or 'Y' key, with the default being off. The size of each individual trail point rendering can be made smaller with the '[' key, and larger with the ']' key.

The color of the active vehicle is by default red and can be altered to a handful of other colors in the ActiveColor sub-menu of the Vehicles pull-down menu. Likewise the inactive color, which is by default yellow, can be altered in the InactiveColor sub-menu. These colors can also be altered by setting the `active_vcolor` and `inactive_vcolor` parameters in the `pMarineViewer` configuration block of the MOOS file. They can be set to any color as described in the Colors Appendix.

10.3.4 The "MOOS-Scope" Pull-Down Menu

The "MOOS-Scope" pull-down menu allows the user to configure the `pMarineViewer` to scope on one or more variables in the MOOSDB. The viewer allows visual scoping on only a single variable at a time, but the user can select different variables via the pull-down menu, or toggle between the current and previous variable with the `'/'` key, or cycle between all registered variables with the `CTRL+'/'` key. The scope fields are on the bottom of the viewer as shown in Figures 24 - 27. The three fields show (a) the variable name, (b) the last time is was updated, and (c) the current value of the variable. Configuration of the menu is done in the MOOS configuration block with entries of the following form:

```
  SCOPE = <variable>, <variable>, ...
```

The keyword `SCOPE` is not case sensitive, but the MOOS variables are. If no entries are provided in the MOOS configuration block, the pull-down menu contains a single item, the `"Add Variable"` item. By selecting this, the user will be prompted to add a new MOOS variable to the scope list. This variable will then immediately become the actively scoped variable, and is added to the pull-down menu.

10.3.5 The Optional "Action" Pull-Down Menu

The "Action" pull-down menu allows the user to invoke pre-define pokes to the MOOSDB (the MOOSDB to which the pMarineViewer is connected). While hooks for a limited number of pokes are available by configuring on-screen buttons (Section 10.5.2), the number of buttons is limited to four. The "Action" pull-down menu allows for as many entries as will reasonably be shown on the screen. Each action, or poke, is given by a variable-value pair, and an optional grouping key. Configuration is done in the MOOS configuration block with entries of the following form:

```
  ACTION = MENU_KEY=<key> # <variable>=<value> # <variable>=<value> # ...
```

If no such entries are provided, this pull-down menu will not appear. The fields to the right of the `ACTION=` are separated by the `'#'` character for convenience to allow several entries on one line. If one wants to use the `'#'` character in one of the variable values, putting double-quotes around the value will suffice to treat the '#' character as part of the value and not the separator. If the pair has the key word MENU_KEY on the left, the value on the right is a key associated with all variable-value pairs on the line. When a menu selection is chosen that contains a key, then all variable-value pairs with that key are posted to the MOOSDB. If the ACTION key word has a trailing '+' character as below, the pull-down menu will render a line separator after the menu item. The following configuration will result in the pull-down menu depicted in Figure 28.

```
ACTION  = MENU_KEY=deploy # DEPLOY = true # RETURN = false
ACTION+ = MENU_KEY=deploy # MOOS_MANUAL_OVERIDE=false
ACTION  = RETURN=true
```

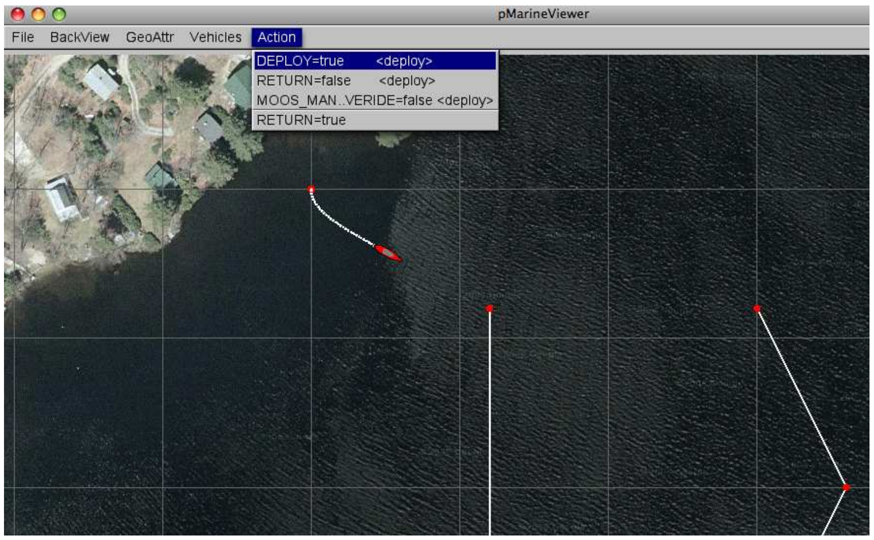


Figure 28: **The Action menu:** The variable value pairs on each menu item may be selected for poking or writing the MOOSDB. The three variable-value pairs above the menu divider will be poked in unison when any of the three are chosen, because they were configured with the same key, `<deploy>`, shown to the right on each item.

The variable-value pair being poked on an action selection will determine the variable type by the following rule of thumb. If the value is non-numerical, e.g., `true`, `one`, it is poked as a string. If it is numerical it is poked as a double value. If one really wants to poke a string of a numerical nature, the addition of quotes around the value will suffice to ensure it will be poked as a string. For example:

```
ACTION   = Vehicle=Nomar # ID="7"
```

As with any other write to the MOOSDB, if a variable has been previously posted with one type, subsequent posts of a different type will be ignored.

10.3.6 The Optional "Mouse-Context" Pull-Down Menu

When the user clicks the left or right mouse in the geo portion of the `pMarineViewer` window, the variables MVIEWER_LCLICK and MVIEWER_RCLICK are published respectively with the geo location of the mouse click, and the name of the active vehicle. This is described in more detail in Section 10.5.1. In short a publication of the following is typical:

```
MVIEWER_LCLICK  = "x=958.0,y=113.0,vname=Unicorn"
```

What happens after this is largely an issue for MOOS processes separate from the `pMarineViewer` application. It is impossible to determine how a user may make use of this. However, the user can configure `pMarineViewer` to augment this published string with additional context that may enable other processes to make better use of this publication. Configuration is done in the MOOS configuration block with entries of the following form:

```
left_context = <context-string>
right_context = <context-string>
```

The `left_context` and `right_context` keywords are case insensitive. If no such entries are provided, this pull-down menu will not appear. The following configuration will result in the pull-down menu depicted in Figure 29.

```
left_context = surface_point
left_context = station_point
left_context = return_point
right_context = loiter_point
```

By selecting `station_point`, for example, the string published to the variable `MVIEWER_LCLICK` would be augmented to look like:

```
MVIEWER_LCLICK  = "x=958.0,y=113.0,vname=alpha,context=station_point"
```

Other MOOS applications could subscribe to this variable and use the information to both alter the state of the helm running on the named vehicle, but also provide the vehicle with a position at which to station-keep.

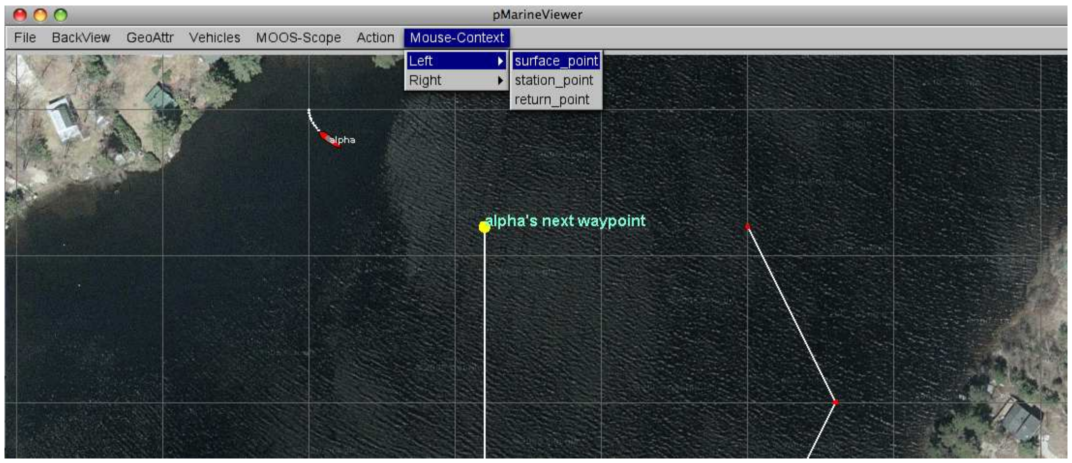


Figure 29: **The Mouse-Context menu:** A string selected from this menu will be appended to the string associated with any mouse click, giving the user a chance to change the context of the click.

10.3.7 The Optional "Reference-Point" Pull-Down Menu

The "Reference-Point" pull-down menu allows the user to select a reference point other than the datum, the `(0,0)` point in local coordinates. The reference point will affect the data displayed in the `Range` and `Bearing` fields in the viewer window. This feature was originally designed for field experiments when vehicles are being operated from a ship. An operator on the ship running the `pMarineViewer` would receive position reports from the unmanned vehicles as well as the present position of the ship. In these cases, the ship is the most useful point of reference. Prior versions of this code would allow for a single declaration of the ship name, but the the current version allows for any number of ship names as a possible reference point. This allows the viewer to be used to display the bearing and range between two deployed unmanned vehicles for example. Configuration is done in the MOOS configuration block with entries of the following form:

```
  reference_vehicle = vehicle
```

If no such entries are provided, this pull-down menu will not appear. When the menu is present, it looks like that shown in Figure 30. When the reference point is a vehicle with a known heading, the user is able to alter the `Bearing` field from reporting either the relative bearing or absolute bearing. Hot keys are defined for each.

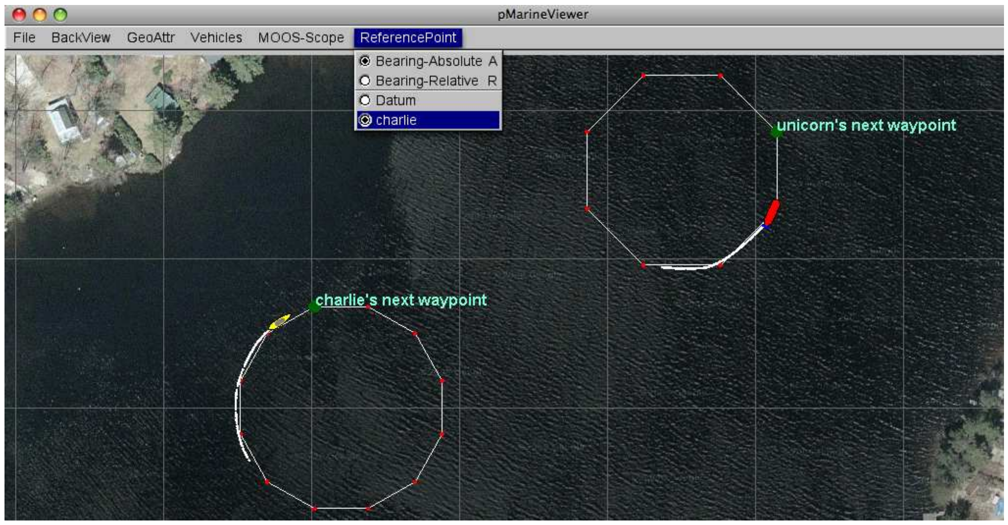


Figure 30: **The Reference-Point menu:** This pull-down menu of the `pMarineViewer` lists the options for selecting a reference point. The reference point determines the values for the `Range` and `Bearing` fields in the viewer for the active vehicle. When the reference point is a vehicle with known heading, the user also may select whether the Bearing is the relative bearing or absolute bearing.

10.4 Displayable Vehicle Shapes, Markers, Drop Points, and other Geometric Objects

The `pMarineViewer` window displays objects in three general categories, (1) the vehicles based on their position reports, (2) markers, which are generally static and things like triangles and squares with labels, and (3) geometric objects such as polygons or lists of line segments that may indicate a vehicle's intended path or other such artifact of it's autonomy situation.

10.4.1 Displayable Vehicle Shapes

The shape rendered for a particular vehicle depends on the *type* of vehicle indicated in the node report received in `pMarineViewer`. There are four types that are currently handled, an AUV shape, a glider shape, a kayak shape, and a ship shape, shown in Figure 31.

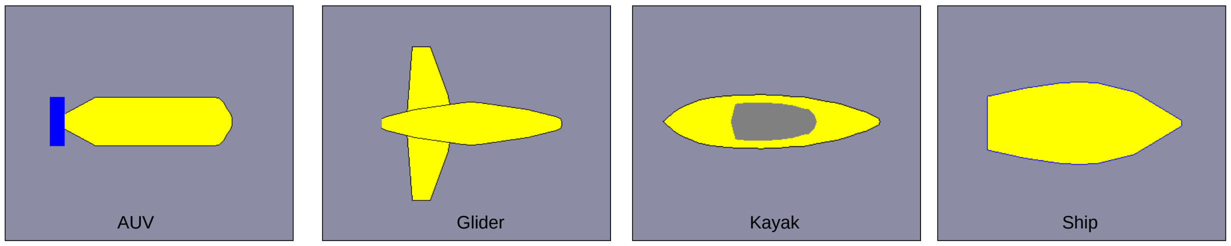


Figure 31: **Vehicles:** Types of vehicle shapes known to the `pMarineViewer`.

The default shape for an unknown vehicle type is currently set to be the shape "ship". The default color for a vehicle is set to be yellow, but can be individually set within the `pMarineViewer` MOOS configuration block with entries like the following:

```
vehicolor = alpha, turquoise
vehicolor = charlie, navy,
vehicolor = philly, 0.5, 0.9, 1.0
```

The parameter `vehicolor` is case insensitive, as is the color name. The vehicle name however is case sensitive. All colors of the form described in the Colors Appendix are acceptable.

10.4.2 Displayable Marker Shapes

A set of simple static markers can be placed on the geo display for rendering characteristics of an operation area such as buoys, fixed sensors, hazards, or other things meaningful to a user. The six types of markers are shown in Figure 32. They are configured in the `pMarineViewer` configuration block of the MOOS file with the following format:

```
// Example marker entries in a pMarineViewer config block of a .moos file
// Parameters are case insensitive. Parameter values (except type and color)
// are case sensitive.
marker  = type=efield,x=100,y=20,label=alpha,COLOR=red,width=4.5
marker  = type=square,lat=42.358,lon=-71.0874,color=blue,width=8
```

Each entry is a string of comma-separated pairs. The order is not significant. The only mandatory fields are for the marker type and position. The position can be given in local x-y coordinates or in earth coordinates. If both are given for some reason, the earth coordinates will take precedent. The `width` parameter is given in meters drawn to scale on the geo display. Shapes are roughly 10x10 meters by default. The GUI provides a hook to scale all markers globally with the 'ALT-M' and 'CTRL-M' hot keys and in the GeoAttributes pull-down menu.

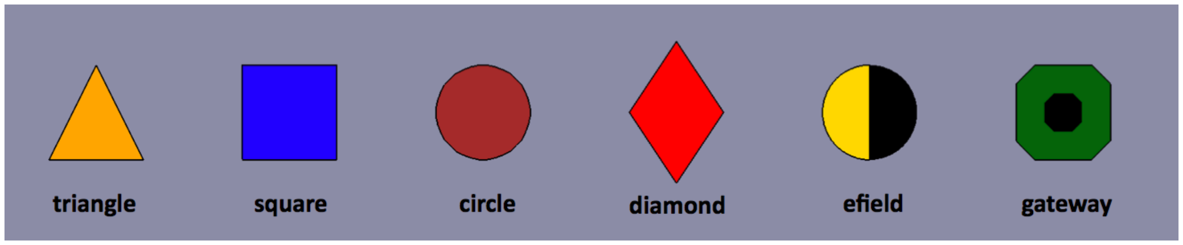


Figure 32: **Markers:** Types of markers known to the `pMarineViewer`.

The color parameter is optional and markers have the default colors shown in Figure 32. Any of the colors described in the Colors Appendix are fair game. The black part of the Gateway and Efield markers is immutable. The label field is optional and is by default the empty string. Note that if two markers of the same type have the same non-empty label, only the first marker will be acknowledged and rendered. Two markers of different types can have the same label.

In addition to declaring markers in the `pMarineViewer` configuration block, markers can be received dynamically by `pMarineViewer` through the `VIEW_MARKER` MOOS variable, and thus can originate from any other process connected to the `MOOSDB`. The syntax is exactly the same, thus the above two markers could be dynamically received as:

```
VIEW_MARKER  = "type=efield,x=100,y=20,SCALE=4.3,label=alpha,COLOR=red,width=4.5"
VIEW_MARKER  = "type=square,lat=42.358,lon=-71.0874,scale=2,color=blue,width=8"
```

The effect of a "moving" marker, or a marker that changes color, can be achieved by repeatedly publishing to the `VIEW_MARKER` variable with only the position or color changing while leaving the label and type the same.

10.4.3 Displayable Drop Points

A user may be interested in determining the coordinates of a point in the geo portion of the `pMarineViewer` window. The mouse may be moved over the window and when holding the `SHIFT` key, the point under the mouse will indicate the coordinates in the local grid. When holding the `CTRL` key, the point under the coordinates are shown in lat/lon coordinates. The coordinates are updated as the mouse moves and disappear thereafter or when the `SHIFT` or `CTRL` keys are release. Drop points may be left on the screen by hitting the left mouse button at any time. The point with coordinates will remain rendered until cleared or toggled off. Each click leaves a new point, as shown in Figure 33.

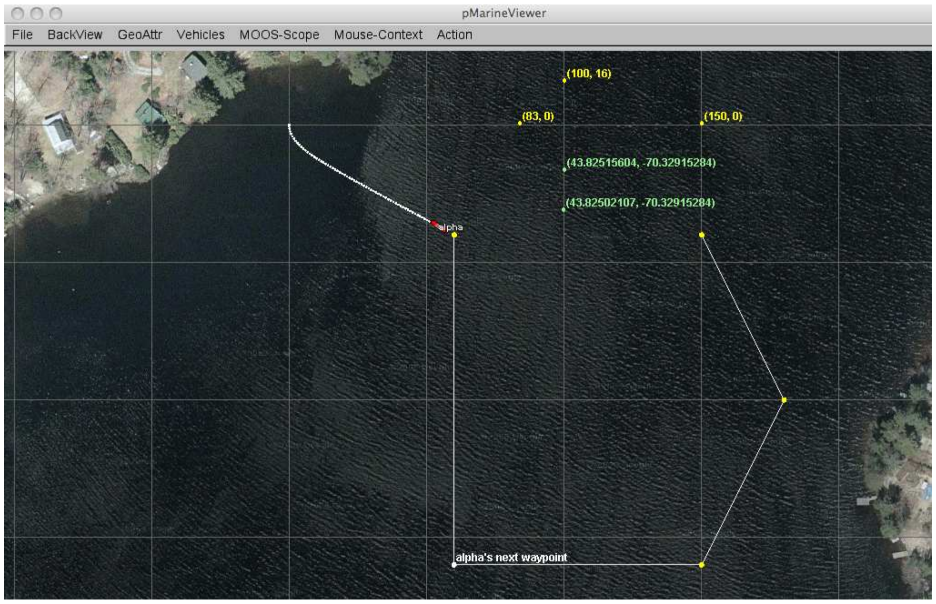


Figure 33: **Drop points:** A user may leave drop points with coordinates on the geo portion of the pMarineViewer window. The points may be rendered in local coordinates or in lat/lon coordinates. The points are added by clicking the left mouse button while holding the `SHIFT` key or `CTRL` key. The rendering of points may be toggled on/off, cleared in their entirety, or reduced by popping the last dropped point.

Parameters regarding drop points are accessible from the `GeoAttr` pull-down menu. The rendering of drop points may be toggled on/off by hitting the 'r' key. The set of drop points may be cleared in its entirety. Or the most recently dropped point may be removed by typing the `CTRL-r` key. The pull-down menu may also be used to change the rendering of coordinates from `"as-dropped"` where some points are in local coordinates and others in lat/lon coordinates, to `"local-grid"` where all coordinates are rendered in the local grid, or `"lat-lon"` where all coordinates are rendered in the lat/lon format.

10.4.4 Displayable Geometric Objects

Some additional objects can be rendered in the viewer such as convex polygons, points, and a set of line segments. In Figures 24 and 25, each vehicle has traversed to and is proceeding around a hexagon pattern. This is apparent from both the rendered hexagon, and confirmed by the trail points. Displaying certain markers in the display can be invaluable in practice to debugging and confirming the autonomy results of vehicles in operation. The intention is to allow for only a few key additional objects to be drawable to avoid letting the viewer become overly specialized and bloated.

In addition to the `NODE_REPORT` variable indicating vehicle pose, `pMarineViewer` registers for the following additional MOOS variables - `VIEW_POLYGON`, `VIEW_SEGLIST`, `VIEW_POINT`. Example values of these variables:

```
VIEW_POLYGON = "label,nyak201-LOITER:85,-9:100,-35:85,-61:55,-61:40,-35:55,-9"
VIEW_POINT   = 10.00,-80.00,5,nyak200
VIEW_SEGLIST = "label,nyak201-WAYPOINT:0,100:50,-35:25,-63"
```

Each variable describes a data structure implemented in the geometry library linked to by pMarineViewer. Instances of these objects are initialized directly by the strings shown above. A key member variable of each geometric object is the *label* since pMarineViewer maintains a (C++, STL) map for each object type, keyed on the label. Thus a newly received polygon replaces an existing polygon with the same label. This allows one source to post its own geometric cues without clashing with another source. By posting empty objects, i.e., a polygon or seglist with zero points, or a point with zero radius, the object is effectively erased from the geo display. The typical intended use is to let a behavior within the helm to post its own cues by setting the label to something unique to the behavior. The VIEW_POLYGON listed above for example was produced by a loiter behavior and describes a hexagon with the six points that follow.

10.5 Support for Command-and-Control Usage

For the most part pMarineViewer is intended to be only a receiver of information from the vehicles and the environment. Adding command and control capability, e.g., widgets to re-deploy or manipulate vehicle missions, can be readily done, but make the tool more specialized, bloated and less relevant to a general set of users. A certain degree of command and control can be accomplished by poking key variables and values into the local MOOSDB, and this section describes three methods supported by pMarineViewer for doing just that.

10.5.1 Poking the MOOSDB with Geo Positions

The graphic interface of pMarineViewer provides an opportunity to poke information to the MOOSDB based on visual feedback of the operation area shown in the geo display. To exploit this, two command and control hooks were implemented with a small footprint. When the user clicks on the geo display, the location in local coordinates is noted and written out to one of two variables - MVIEWER_LCLICK for left mouse clicks, and MVIEWER_RCLICK for right mouse clicks, with the following syntax:

```
MVIEWER_LCLICK  = "x=958.0,y=113.0,vname=nyak200",
```

and

```
MVIEWER_RCLICK  = "x=740.0,y=-643.0,vname=nyak200".
```

One can then write another specialized process, e.g., pViewerRelay, that subscribes to these two variables and takes whatever command and control actions desired for the user's needs. One such incarnation of pViewerRelay was written (but not distributed or addressed here) that interpreted the left mouse click to have the vehicle station-keep at the clicked location.

10.5.2 Configuring GUI Buttons for Command and Control

The pMarineViewer GUI can be optionally configured to allow for four push-buttons to be enabled and rendered in the lower-right corner. Each button can be associated with a button label, and a list of variable-value pairs that will be poked to the MOOSDB to which the pMarineViewer process is connected. The basic syntax is as follows:

```
BUTTON_ONE   = <label> # <variable>=VALUE # <variable>=<value>  ...
BUTTON_TWO   = <label> # <variable>=VALUE # <variable>=<value>  ...
BUTTON_THREE = <label> # <variable>=VALUE # <variable>=<value>  ...
BUTTON_FOUR  = <label> # <variable>=VALUE # <variable>=<value>  ...
```

The left-hand side contains one of the four button keywords, e.g., `BUTTON_ONE`. The right-hand side consists of a `'#'`-separated list. Each component in this list is either a `'='`-separated variable-value pair, or otherwise it is interpreted as the button's label. The ordering does not matter and the `'#'`-separated list can be continued over multiple lines as in lines 59-60 in Listing 18 on page 111.

The variable-value pair being poked on a button call will determine the variable type by the following rule of thumb. If the value is non-numerical, e.g., `true`, `one`, it is poked as a string. If it is numerical it is poked as a double value. If one really wants to poke a string of a numerical nature, the addition of quotes around the value will suffice to ensure it will be poked as a string. For example:

```
BUTTON_ONE   = Start # Vehicle=Nomar # ID="7"
```

In this case, clicking the button labeled `"Start"` will result in two pokes, the second of which will have a string value of `"7"`, not a numerical value. As with any poke to the `MOOSDB` of a given variable-value pair, if the value is of a type inconsistent with the first write to the DB under that variable name, it will simply by ignored.

As described in Section 10.3.5, additional variable-value pairs for poking the MOOSDB can be configured in the "Action" pull-down menu. Unlike the use of buttons, which is limited to four, the number of actions in the pull-down menu is limited only by what can reasonably be rendered on the user's screen.

10.6 Configuration Parameters for pMarineViewer

Many of the display settings available in the pull-down menus described in Sections 10.3 can also be set in the `pMarineViewer` block of the MOOS configuration file. Mostly this redundancy is for convenience for a user to have the desired settings without further keystrokes after start-up. An example configuration block is shown in Listing 18.

Parameter	**Description**	**Allowed Values**	**Default**
`hash_view`	Turning off or on the hash marks.	`true, false`	`true`
`hash_delta`	Distance between hash marks	`[10, 1000]`	`50`
`hash_shade`	Shade of hash marks - 0 is black to 1 is white	`[0, 1.0]`	`0.65`
`back_shade`	Shade of hash marks - 0 is black to 1 is white	`[0, 1.0]`	`0.55`
`tiff_view`	Background image used if set to true	`true, false`	`true`
`tiff_type`	Uses the first (A) image if set to `true`	`true, false`	`true`
`tiff_file`	Filename of a tiff file background image	any tiff file	`Default.tif`
`tiff_file_b`	Filename of a tiff file background image	any tiff file	`DefaultB.tif`
`view_center`	The center of the viewing image (the zoom-to point)	`(x, y)`	`(0, 0)`

Table 6: **Background parameters:** Parameters affecting the rendering of the `pMarineViewer` background.

Parameter	Description	Allowed Values	Default
`trails_color`	Color of points rendered in a trail history	`any color`	`white`
`trails_connect_viewable`	Render lines between dots if true	`true, false`	`false`
`trails_history_size`	Number of points stored in a trail history	`[0, 10,000]`	`1,000`
`trails_length`	Number of points rendered in a trail history	`[0, 10,000]`	`100`
`trails_point_size`	Size of dots rendered in a trail history	`[0, 100]`	`1`
`trails_viewable`	Trail histories note rendered if false	`true, false`	`true`
`vehicles_active_color`	Color of the one active vehicle	`any color`	`red`
`vehicles_inactive_color`	Color of other inactive vehicles	`any color`	`yellow`
`vehicles_name_active`	Active vehicle set to the named vehicle	`known name`	`1st`
`vehicles_name_center`	Center vehicle set to the named vehicle	`known name`	`n/a`
`vehicles_name_color`	Color of the font for all vehicle labels	`any color`	`white`
`vehicles_name_viewable`	Vehicle labels not rendered if set to off	`off, names, names+mode, names+depth`	`names`
`vehicles_shape_scale`	Change size rendering - 1.0 is actual size	`[0.1, 100]`	`1`
`vehicles_viewable`	Vehicles not rendered is set to false	`true, false`	`false`
`vehicolor`	Override inactive vehicle color individually	`See p.x`	`n/a`

Table 7: **Vehicle parameters:** Parameters affecting how vehicles are rendered in `pMarineViewer`.

Parameter	Description	Allowed Values	Default
circle_edge_color	Color rendered circle lines	any color	yellow
circle_edge_width	Line width of rendered circle lines	[0, 10]	2
grid_edge_color	Color of rendered grid lines	any color	white
grid_edge_width	Line width of rendered grid lines	[0, 10]	2
grid_viewable_all	If true grids will be rendered	true, false	true
grid_viewable_labels	If true grid labels will be rendered	true, false	true
point_viewable_all	If true points will be rendered	true, false	true
point_viewable_labels	If true point labels will be rendered	true, false	true
point_vertex_color	Color of rendered points	any color	yellow
point_vertex_size	Size of rendered points	[0, 10]	4
polygon_edge_color	Color of rendered polygon lines	any color	yellow
polygon_edge_width	Line width of rendered polygon edges	[0, 10]	1
polygon_label_color	Color rendered polygon labels	any color	khaki
polygon_viewable_all	If true all polygons are rendered	true, false	true
polygon_viewable_labels	If true polygon labels are rendered	true, false	true
polygon_vertex_color	Color of rendered polygon vertices	any color	red
polygon_vertex_size	Size of rendered polygon vertices	[0, 10]	3
seglist_edge_color	Color or rendered seglist lines	any color	white
seglist_edge_width	Line width of rendered seglist edges	[0,10	1
seglist_label_color	Color of rendered seglist labels	any color	orange
seglist_viewable_all	If true all seglists are rendered	true, false	true
seglist_viewable_labels	If true seglist labels are rendered	true, false	true
seglist_vertex_color	Color of rendered seglist vertices	any color	blue
seglist_vertex_size	Size of rendered seglist vertices	[0, 10]	3

Table 8: **Geometric parameters:** Parameters affecting the rendering of the pMarineViewer geometric objects.

Parameter	Description	Allowed Values	Default
marker	Add and newly defined marker	See p. 105	n/a
markers_viewable	If true all markers are rendered	true, false	true
markers_labels_viewable	If true marker labels are rendered	true, false	true
markers_scale_global	Marker widths are multiplied by this factor	[0.1, 100	1
markers_label_color	Color of rendered marker labels	any color	white

Table 9: **Marker parameters:** Parameters affecting the rendering of the pMarineViewer markers.

Listing 18 - An example pMarineViewer configuration block.

```
LatOrigin  =   47.7319
LongOrigin = -122.8500

//------------------------------------------------
// pMarineViewer configuration  block

ProcessConfig = pMarineViewer
{
  // Standard MOOS parameters affecting comms and execution
```

```
AppTick    = 4
CommsTick  = 4

// Set the background images
TIFF_FILE   = long_beach_sat.tif
TIFF_FILE_B = long_beach_map.tif


// Parameters and their default values
hash_view     = false
hash_delta    = 50
hash_shade    = 0.65
back_shade    = 0.70
trail_view    = true
trail_size    = 0.1
tiff_view     = true
tiff_type     = true
zoom          = 1.0
vehicles_name_viewable = false

// Setting the vehicle colors - default is yellow
vehicolor     = henry,dark_blue
vehicolor     = ike,0.0,0.0,0.545
vehicolor     = jane,hex:00,00,8b

// All polygon parameters are optional - defaults are shown
// They can also be set dynamically in the GUI in the GeoAttrs pull-down menu
polygon_edge_color   = yellow
polygon_vertex_color = red
polygon_label_color  = khaki
polygon_edge_width   = 1.0
polygon_vertex_size  = 3.0
polygon_viewable_all = true;
polygon_viewable_labels = true;

// All seglist parameters are optional - defaults are shown
// They can also be set dynamically in the GUI in the GeoAttrs pull-down menu
seglist_edge_color   = white
seglist_vertex_color = dark_blue
seglist_label_color  = orange
seglist_edge_width   = 1.0
seglist_vertex_size  = 3.0
seglist_viewable_all = true;
seglist_viewable_labels = true;

// All point parameters are optional - defaults are shown
// They can also be set dynamically in the GUI in the GeoAttrs pull-down menu
point_vertex_size  = 4.0;
point_vertex_color = yellow
point_viewable_all = true;
point_viewable_labels = true;

// Define two on-screen buttons with poke values
button_one   = DEPLOY # DEPLOY=true
button_two   = MOOS_MANUAL_OVERIDE=false # RETURN=false
```

```
64    button_two   = RETURN # RETURN=true
65    button_three = DEPTH-10 # OPERATION_DEPTH=10
66    button_four  = DEPTH-30 # OPERATION_DEPTH=30
67
68    // Declare variable for scoping. Variable names case sensitive
69    scope = PROC_WATCH_SUMMARY
70    scope = BHV_WARNING
71    scope = BHV_ERROR
72
73    // Declare Variable-Value pairs for convenient poking of the MOOSDB
74    action = OPERATION_DEPTH=50
75    action = OPERATION_DEPTH=0 # STATUS="Coming To the Surface"
76  }
```

Color references as in lines 27-29 can be made by name or by hexadecimal or decimal notation. (All three colors in lines 27-29 are the same but just specified differently.) See the Colors Appendix for a list of available color names and their hexadecimal equivalent.

The `VERBOSE` parameter on line 24 controls the output to the console. The console output lists the types of mail received on each iteration of `pMarineViewer`. In the non-verbose mode, a single character is output for each received mail message, with a '*' for `NODE_REPORT`, a 'P' for a `VIEW_POLYGON`, a '.' for a `VIEW_POINT`, and a 'S' for a `VIEW_SEGLIST`. In the verbose mode, each received piece of mail is listed on a separate line and the source of the mail is also indicated. An example of both modes is shown in Listing 19.

Listing 19 - An example pMarineViewer console output.

```
// Example pMarineViewer console output NOT in verbose mode

13.56 > ****..
13.82 > **..
14.08 > **..
14.35 > **..
14.61 > ****.P.P
14.88 > **..
15.14 > **..

// Example pMarineViewer console output in verbose mode

15.42 >
   NODE-REPORT(nyak201)
   NODE-REPORT(nyak200)
   Point(nyak201_wpt)
   Point(nyak200_wpt)

15.59 >
   Point(nyak201)
   Poly(nyak201-LOITER)
   NODE-REPORT(nyak201)
   NODE-REPORT(nyak200)
   Point(nyak200)
   Poly(nyak200-LOITER)
```

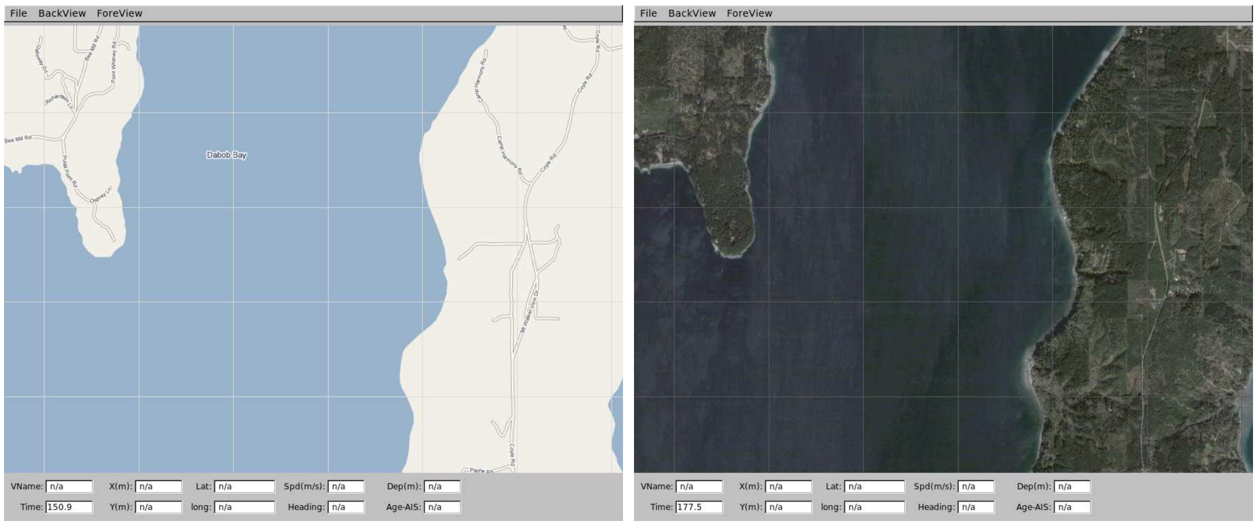
10.7 More about Geo Display Background Images

The geo display portion of the viewer can operate in one of two modes, a grey-scale background, or an image background. Section 10.3.1 addressed how to switch between modes in the GUI interface. To use an image in the geo display, the input to `pMarineViewer` comes in two files, an image file in TIFF format, and an information text file correlating the image to the local coordinate system. The file names should be identical except for the suffix. For example `dabob_bay.tif` and `dabob_bay.info`. Only the `.tif` file is specified in the `pMarineViewer` configuration block of the MOOS file, and the application then looks for the corresponding `.info` file. The info file contains six lines - an example is given in Listing 20.

Listing 20 - An example .info file for the pMarineViewer

```
// Lines may be in any order, blank lines are ok
// Comments begin with double slashes

datum_lat  = 47.731900
datum_lon  = -122.85000
lat_north  = 47.768868
lat_south  = 47.709761
lon_west   = -122.882080
lon_east   = -122.794189
```

All six parameters are mandatory. The two datum lines indicate where $(0,0)$ in local coordinates is in earth coordinates. The `lat_north` parameters correlates the upper edge of the image with its latitude position. Likewise for the other three parameters and boundaries. Two image files may be specified in the `pMarineViewer` configuration block. This allows a map-like image and a satellite-like image to be used interchangeably during use. (Recall the ToggleBackGroundType entry in the BackView pull-down menu discussed earlier.) An example of this is shown in Figure 34 with two images of Dabob Bay in Washington State. Both image files where created from resources at www.maps.google.com.



VName: n/a X(m): n/a Lat: n/a Spd(m/s): n/a Dep(m): n/a
Time: 150.9 Y(m): n/a long: n/a Heading: n/a Age-AIS: n/a

VName: n/a X(m): n/a Lat: n/a Spd(m/s): n/a Dep(m): n/a
Time: 177.5 Y(m): n/a long: n/a Heading: n/a Age-AIS: n/a

Figure 34: **Dual background geo images:** Two images loaded for use in the geo display mode of `pMarineViewer`. The user can toggle between both as desired during operation.

In the configuration block, the images can be specified by:

```
TIFF_FILE   = dabob_bay_map.tif
TIFF_FILE_B = dabob_bay_sat.tif
```

By default `pMarineViewer` will look for the files `Default.tif` and `DefaultB.tif` in the local directory unless alternatives are provided in the configuration block.

10.8 Publications and Subscriptions for pMarineViewer

10.8.1 Variables published by the pMarineViewer application

Variables published by `pMarineViewer` are summarized in Table 10 below. A more detail description of each variable follows the table.

#	Variable	Description
1	`MVIEWER_LCLICK`	The position in local coordinates of a user left mouse button click
2	`MVIEWER_RCLICK`	The position in local coordinates of a user right mouse button click
3	`HELM_MAP_CLEAR`	.

Table 10: **Variables published by the `pMarineViewer` application.**

- `MVIEWER_LCLICK`: When the user clicks the left mouse button, the position in local coordinates, along with the name of the active vehicle is reported. This can be used as a command and control hook as described in Section 10.5. As an example:

  ```
  MVIEWER_LCLICK = ‘‘x=-56.0,y=-110.0,vname=alpha’’
  ```

- `MVIEWER_RCLICK`: This variable is published when the user clicks with the right mouse button. The same information is published as with the left click.
- `HELM_MAP_CLEAR`: This variable is published once when the viewer connects to the `MOOSDB`. It is used in the `pHelmIvP` application to clear a local buffer used to prevent successive identical publications to its variables.

10.8.2 Variables subscribed for by pMarineViewer application

- `NODE_REPORT`: This is the primary variable consumed by `pMarineViewer` for collecting vehicle position information. An example:

```
NODE_REPORT = "NAME=nyak201,TYPE=kayak,MOOSDB_TIME=53.049,UTC_TIME=1195844687.236,X=37.49,
              Y=-47.36, SPD=2.40,HDG=11.17,DEPTH=0"
```

- `NODE_REPORT_LOCAL`: This serves the same purpose as the above variable. In some simulation cases this variable is used.

- `VIEW_POLYGON`: A string representation of a polygon.
- `VIEW_POINT`: A string representation of a point.
- `VIEW_SEGLIST`: A string representation of a segment list.
- `TRAIL_RESET`: When the viewer receives this variable it will clear the history of trail points associated with each vehicle. This is used when the viewer is run with a simulator and the vehicle position is reset and the trails become discontinuous.
- `GRID_CONFIG`: A string representation of a grid. This initializes and registers a new grid with the viewer.
- `GRID_DELTA`: A string representation of a change in values for a given grid and specific grid cells with new value for each given cell.

11 Behaviors of the IvP Helm

The following is a description of some single-vehicle behaviors currently written for the IvP Helm. The division of single-vehicle behaviors and multi-vehicle behaviors (next section) is somewhat arbitrary. Other behavior modules exist that may be either in a testing state or too specific to a project for discussion here. The below description is for the person who wants to *use* current behaviors in the toolbox. The topic of how to add a new behavior is not covered here.

A behavior has a standard parameters defined at the `IvPBehavior` level as well as unique parameters defined at the subclass level. Parameters are set in the behavior file. For a behavior user, the setting of parameters is the primary venue 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:

```
parameter = 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.

11.1 BHV_Waypoint

11.1.1 Overview of the BHV_Waypoint Behavior

The `BHV_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 35.

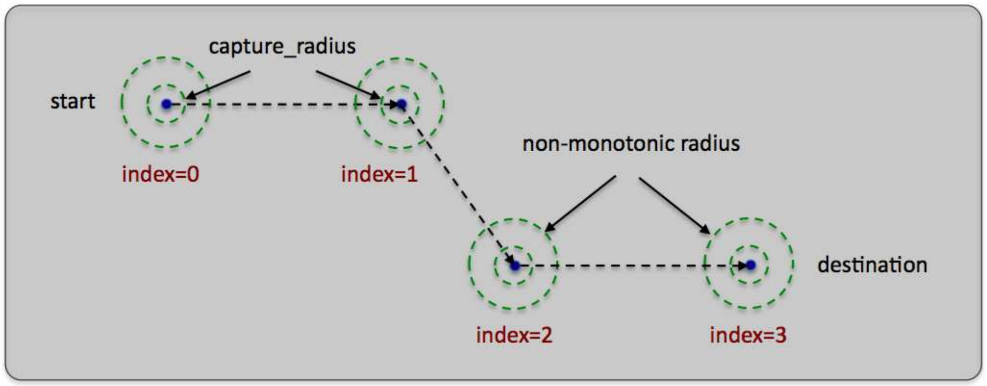


Figure 35: **The `BHV_Waypoint` behavior:** The waypoint behavior basic purpose is to traverse a set of waypoints. A capture radius is specified to define what it means to have achieved a waypoint, and a non-monotonic radius is specified to define what it means to be "close enough" should progress toward the waypoint be noted to degrade.

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.

11.1.2 Brief Summary of the BHV_Waypoint Behavior Parameters

The following parameters are defined for this behavior. A more detailed description is provided other parts of this section, and in Table 11 on page 147.

`POINTS`:	A colon separated list of x,y pairs given as points in 2D space, in meters.
`POLYGON`:	An alias for `POINTS`.
`SPEED`:	The desired speed (m/s) at which the vehicle travels through the points.
`CAPTURE_RADIUS`:	The radius tolerance, in meters, for satisfying the arrival at a waypoint.
`RADIUS`:	An alias for `CAPTURE_RADIUS`.
`NM_RADIUS`:	An "outer" capture radius. Arrival declared when the vehicle is in this range and the distance to the next waypoint begins to increase.
`ORDER`:	The order in which waypoints are traversed - `"normal"`, or `"reverse"`.
`LEAD`:	If this parameter is set, track-line following between waypoints is enabled.
`LEAD_DAMPER`:	Distance from trackline within which the `lead` distance is stretched out.
`REPEAT`:	The number of *extra* times traversed through the waypoints.
`WPT_STATUS_VAR`:	The MOOS variable posting a status report. The default is `WPT_STAT`.
`WPT_INDEX_VAR`:	The MOOS variable posting the index of the behavior's next waypoint.
`CYCLE_FLAGS`:	MOOS variable-value pairs posted at end of each cycle through waypoints.
`CYCLE_INDEX_VAR`:	The MOOS variable posting # of times cycled through the waypoints.
`POST_SUFFIX`:	A suffix tagged onto the `WPT_STATUS`, `WPT_INDEX` and `CYCLE_INDEX` variables.

11.1.3 Specifying Waypoints - the `points`, `order`, and `repeat` Parameters

The waypoints may be specified explicitly as a colon-separated list of comma-separate pairs, or implicitly using a geometric description. The order of the parameters may also be reversed with the `order` parameter. An example specification:

```
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 not continue to launch. If the syntactic error is passed as part of a dynamic update (see

Section 7.2.2), the change in waypoints will be ignored and the a warning posted to the `BHV_WARNING` variable. See Section 9 for more methods for specifying sets of waypoints.

11.1.4 The `capture_radius` and `nonmonotonic_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 35. The non-monotonic radius or `nm_radius` parameter specifies an alternative criteria for achieving a waypoint.

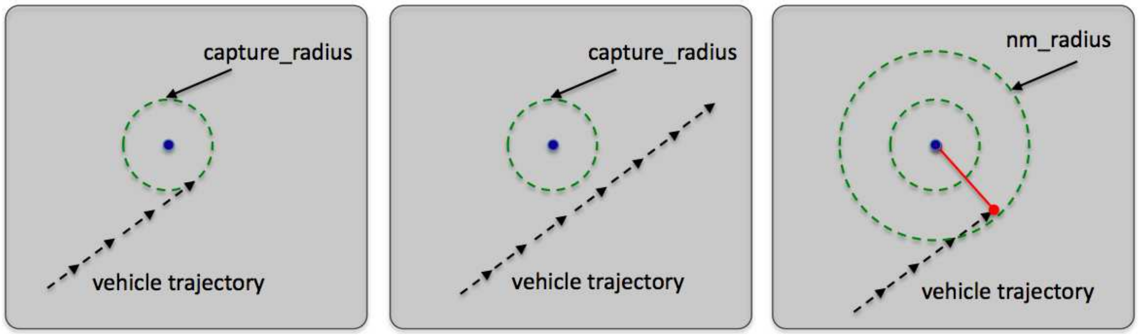


Figure 36: **The capture radius and non-monotonic radius:** (a) a successful waypoint arrival by achieving proximity less than the capture radius. (b) a missed waypoint likely resulting in the vehicle looping back to try again. (c) a missed waypoint but arrival declared anyway when the distance to the waypoint begins to increase and the vehicle is within the non-monotonic 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 capture radius. The `nm_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 36. The behavior keeps a running tally of hits achieved with the capture radius and those achieved with the non-monotonic radius. These tallies are reported in a status message described in Section 11.1.6 below.

11.1.5 Track-line Following using the `lead` Parameter

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.

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

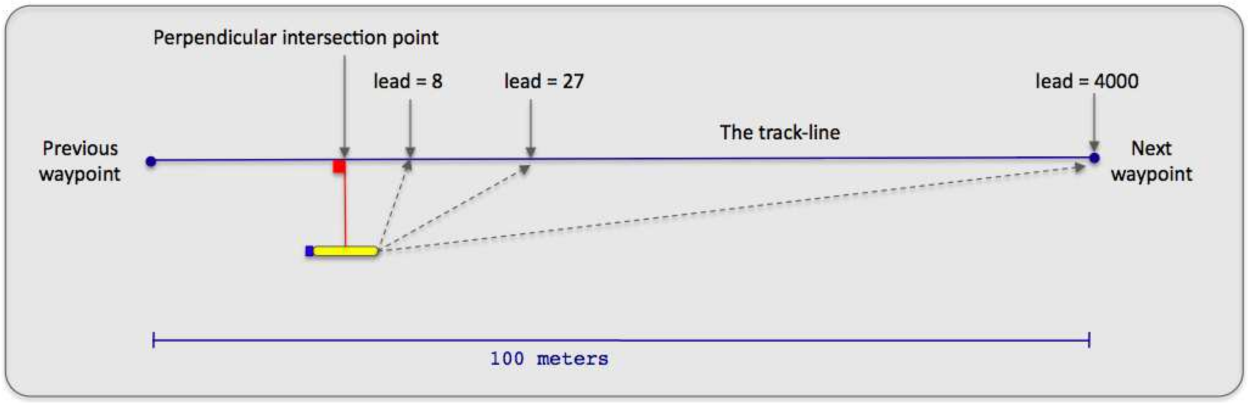


Figure 37: **The track-line mode:** When in track line mode, the vehicle steers toward a point o the track line rather than simply toward the next waypoint. The steering-point is determined by the `lead` parameter. This is the distance from the perpendicular intersection point toward the next waypoint.

the 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.

If the `lead` parameter is enabled, it may be optionally used in conjuction 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.

11.1.6 Variables Published by the BHV_Waypoint Behavior

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

```
WPT_STAT = "vname=alpha,behavior=traverse1,index=0,dist=43,eta=23"
```

The `WPT_INDEX` variable simply publishes the index of the current waypoint. This is a bit redundant, 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, or suppress it completely:

```
wpt_status_var  = MY_WPT_STATUS_VAR      // The default is "WPT_STAT"
wpt_index_var   = MY_WPT_INDEX_VAR       // The default is "WPT_INDEX"
cycle_index_var = MY_CYCLE_INDEX_VAR     // The default is "CYCLE_INDEX"
```

or, to suppress the reports:

```
wpt_status_var  = silent  // both case insensitive
wpt_index_var   = silent
cycle_index_var = 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 `cycleflags` 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` 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 6 on page 32.

11.1.7 The Objective Function Produced by the BHV_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.

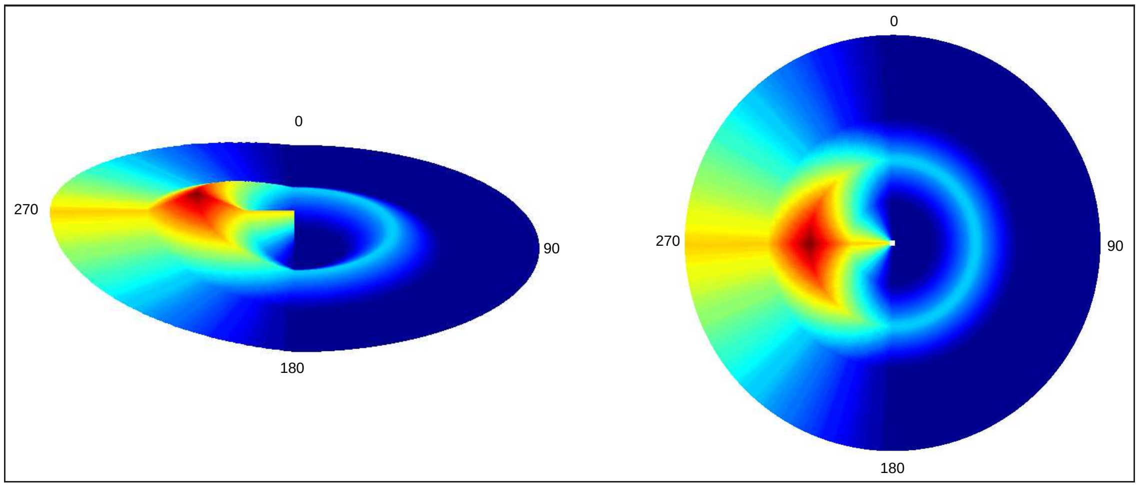


Figure 38: **A waypoint objective function:** The objective function produced by the waypoint behavior is defined over possible heading and speed values. Depicted here is an objective function favoring maneuvers to a waypoint 270 degrees from the current vehicle position and favoring speeds closer to the mid-range of capable vehicle speeds. Higher speeds are represented farther radially out from the center.

11.2 BHV_OpRegion

11.2.1 Overview of the BHV_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.

11.2.2 Brief Summary of the BHV_OpRegion Parameters

The following parameters are defined for this behavior. A more detailed description is provided other parts of this section, and in Table 12 on page 148.

`POLYGON:`	The lat-lon area the vehicle is restricted to stay within. Section 11.2.3.
`TRIGGER_ENTRY_TIME:`	The time required for the vehicle to have been within the polygon region before triggering the polygon requirement. Section 11.2.3.
`TRIGGER_EXIT_TIME:`	The time required to have been outside the polygon before declaring a polygon containment failure. Section 11.2.3.
`MAX_TIME:`	The max allowable time in seconds. Section 11.2.4.
`MAX_DEPTH:`	The max allowable depth in meters. Section 11.2.5.
`MIN_ALTITUDE:`	The min allowable altitude in meters. Section 11.2.6.

11.2.3 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.

11.2.4 Safety Checking Applied to a Maximum Mission Operation Time

`MAX_TIME:` The maximum allowable time (in seconds) that the helm is allowed run. The clock starts when the pHelmIvP process first takes control.

11.2.5 Safety Checking Applied to a Maximum Vehicle Depth

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

11.2.6 Safety Checking Applied to a Minimum Vehicle Altitude

`MIN_ALTITUDE:` The minimum allowable altitude of the vehicle (in meters). If no depth is provided, no depth checks are made.

11.2.7 Variables Published by the BHV_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 39.

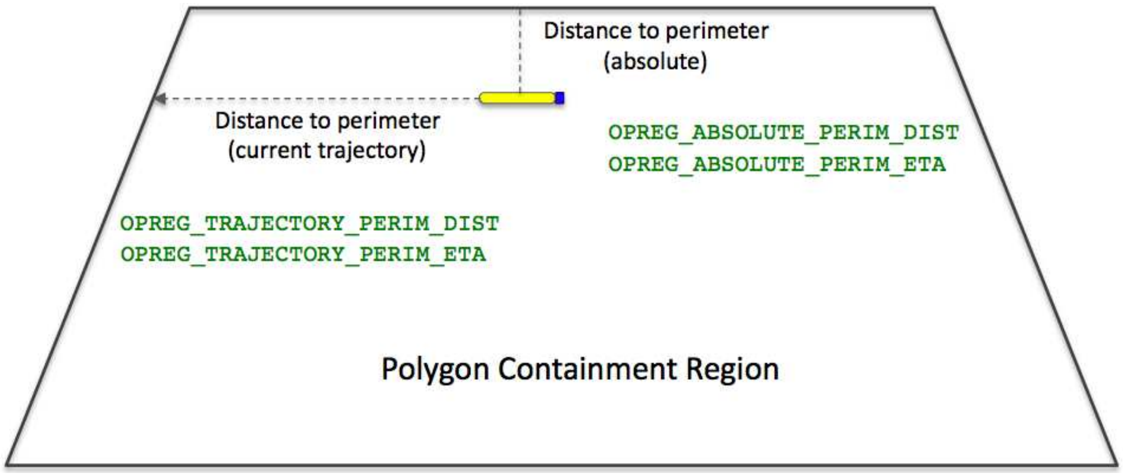

Distance to perimeter (absolute)
Distance to perimeter (current trajectory)
OPREG_ABSOLUTE_PERIM_DIST
OPREG_ABSOLUTE_PERIM_ETA
OPREG_TRAJECTORY_PERIM_DIST
OPREG_TRAJECTORY_PERIM_ETA
Polygon Containment Region

Figure 39: **The OpRegion polygon and status variables:** The OpRegion behavior publishes information regarding its estimated distance and time of arrival (ETA) to the perimeter of the polygon containment region. It publishes two sets of information; one based on the current trajectory and the other based on the absolute distance to the perimeter at top vehicle speed.

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.

11.3 BHV_Loiter

A behavior for transiting to and repeatedly traversing a set of waypoints. A similar effect can be achieved with the BHV_Waypoint behavior but this behavior assumes a set of waypoints forming a convex polygon to exploit certain useful algorithms discussed below. This behavior is comparable to the "Obit Task" of the older helm but is more general in that general convex polygons, not just those approximating circles, are allowed. It also utilizes the non-monotonic arrival criteria use in the BHV_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. And 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. The following parameters are defined for this behavior:

`POLYGON:` A colon separated list of comma-separated x,y pairs indicating points in 2D space. Units are in in meters. Unlike the waypoint behavior, these points must describe a convex polygon; if the convexity condition fails the behavior will not instantiate. As an alternative to listing a sequence of points, a orbit-style polygon can be given by four values (1),(2) the x and y position, (3) the radius in meters, and (4) the number of points on the circle. This specification is denoted with the "radial" tag as follows "radial:50,50,200,16".

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

`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.

`NM_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 `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 36). As a rule of thumb, a distance of twice the arrival radius is practical.

`CLOCKWISE:` If "true", the behavior will influence the vehicle in a clockwise direction around the polygon. Values are case insensitive, but must spell either true or false. The default is true.

`ACQUIRE_DIST:` 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.)

`POST_SUFFIX:` This string will 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 variabes 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.

When the behavior is active, it is in either one of two modes; the *acquire* mode or *normal* mode. In the normal mode it is merely proceeding to the next waypoint on the polygon. In the acquire mode, each iteration begins by first determining the next polygon point to treat as the next waypoint. This is useful for ensuring the entry waypoint isn't followed by a need for a sharp vehicle turn. The acquire point depends on the chosen direction of polygon traversal, as shown in Figure 40.

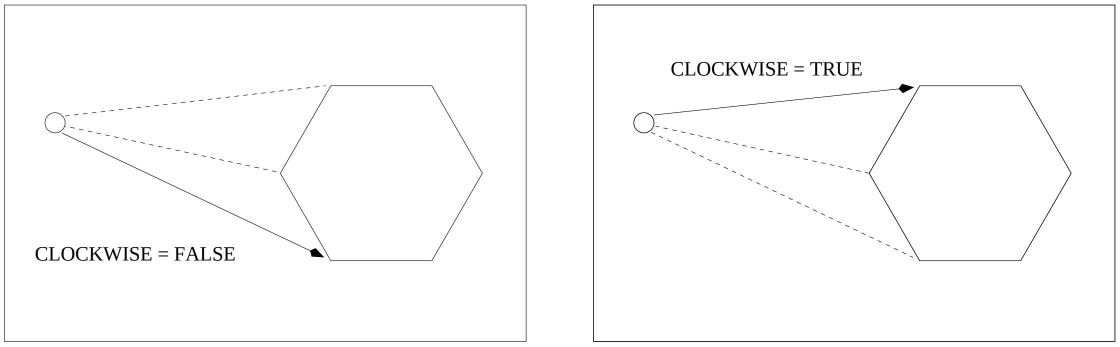


Figure 40: In the *acquire* mode, the polygon points are evaluated for suitability in terms of a smooth entry trajectory. Only the "viewable" points, those viewable if the polygon were an opaque object and the viewer were at the current vehicle location, are contenders. The contenders are rated on the follow-on angle given the desired clockwise or counter-clockwise loiter direction. Larger follow-on angles are preferred as shown.

When the behavior is in the acquire mode and *outside* the polygon, the chosen vertex is the one most tangential in either the clockwise or counter-clockwise direction as shown in the figure. When the vehicle is *inside* the polygon, the chosen vertex is the one which forms the most obtuse angle between the current vehicle position, the vertex, and the follow-on vertex. Unlike the case when outside the polygon, the chosen vertex changes as the vehicle makes progress back to the polygon perimeter. The effect is for the vehicle to "spiral" out to the perimeter for the smoothest re-entry in to a normal loitering path.

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.

11.4 BHV_PeriodicSpeed

This behavior will periodically influence the speed of the vehicle while remaining neutral at other times. The timing is specified by a given period length in which the influence is on, and a gap length specifying the time between periods. 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. The following parameters are defined for this behavior:

`PERIOD_LENGTH:` The duration of the period, in seconds, during which the behavior will produce an objective function over the desired speed.

`PERIOD_GAP:` The duration of time in seconds between periods.

`PERIOD_FLAG:` A flag (MOOS variable) to be posted at the beginning of each active period. The argument is of the form `VAR=VAL`. If if no value is specified, the value will be the period index, incremented on each new period commencement.

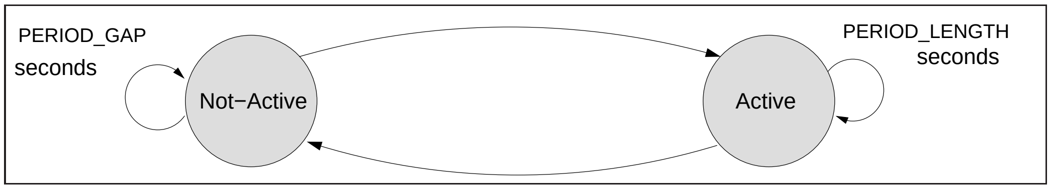


Figure 41: In active mode the behavior will produce an objective function defined over speed that will potentially influence the speed of the vehicle. In the inactive mode, it simply will not produce an objective function.

`STAT_PENDING_ACTIVE:` The number of seconds remaining until the behavior reaches the *active* state. By default this is empty and no status is posted by the behavior. To reduce posting volume, the value posted will be rounded to the nearest second until less than one second remains in which case fractions are posted.

`STAT_PENDING_INACTIVE:` The number of seconds remaining until the behavior reaches the *inactive* state. By default this is empty and no status is posted by the behavior. To reduce posting volume, the value posted will be rounded to the nearest second until less than one second remains in which case fractions are posted.

`PERIOD_SPEED:` The desired speed in meters per second.

`PERIOD_PEAKWIDTH:` The width of the peak in meters per second in the speed objective function.

`PERIOD_BASEWIDTH:` The width of the base, in meters per second in the speed objective function.

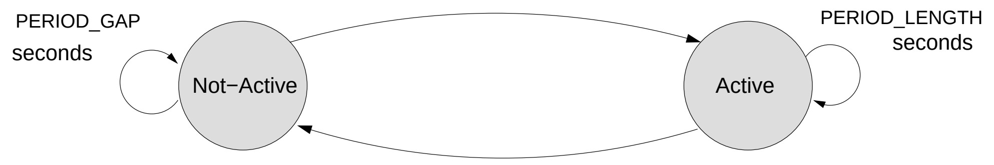


Figure 42: In (a) the preference is a for a particular speed and a slight tolerance in either direction. In (b) the preference is for a particular range of speeds with a slight tolerance either way. In (c) the preference is for anything less than a given speed with some tolerance for higher speeds. In (d) the preference is for anything greater than a given speed with a no tolerance for lower speeds.

11.5 BHV_PeriodicSurface

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 specified 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. The behavior can be in one of four states as described in Figure 43 below.

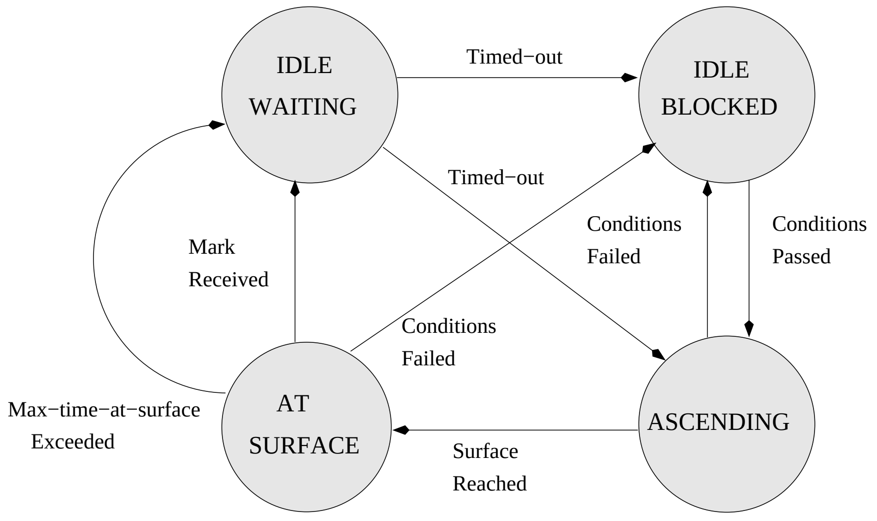


Figure 43: Possible modes of the PeriodicSurface behavior.

In the `IDLE_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 `IDLE_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 `IDLE_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.

`PERIOD:` The duration of the period, in seconds, during which the behavior will remain in the `IDLE_WAITING` state.

`MARK_VARIABLE:` The name of 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.

`PENDING_STATUS_VAR:` This variable will 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.

`ATSURFACE_STATUS_VAR:` This variable will be written to with the number of seconds that the vehicle has been waiting at the surface (for the event indicated by the `MARK_VARIABLE`). 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 argument names a MOOS variable, it is case sensitive.

`ASCENT_SPEED:` This 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.

`ASCENT_GRADE:` This 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*. 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.

`ZERO_SPEED_DEPTH:` The depth (in meters) during the ascent state at which the desired speed becomes zero, and presumably further ascent is achieved through positive buoyancy.

`MAX_TIME_AT_SURFACE:` The maximum time (in seconds) spent in the `AT_SURFACE` state, waiting for the event indicated by the `MARK_VARIABLE`, before the behavior transitions into the `IDLE` state.

11.6 BHV_ConstantDepth

This behavior will drive the vehicle at a specified depth. Analogous to the ConstantDepthTask in the pHelm task library, but somewhat different. 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:

`DEPTH:` The desired depth in meters.

`PEAKWIDTH:` The width of the peak in meters in the produced objective function.

`BASEWIDTH:` The width of the base, in meters in the produced objective function.

`DURATION:` This is a 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.

11.7 BHV_ConstantHeading

This behavior will drive the vehicle at a specified depth. Analogous to the ConstantHeadingTask in the pHelm task library, but somewhat different. 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:

`HEADING:` The desired heading in degrees (-180, +180].

`PEAKWIDTH:` The width of the peak in degrees in the produced objective function.

`BASEWIDTH:` The width of the base, in degrees in the produced objective function.

`DURATION:` This is a 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.

11.8 BHV_ConstantSpeed

This behavior will drive the vehicle at a specified speed. Analogous to the ConstantSpeedTask in the pHelm task library, but somewhat different. 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:

`SPEED:` The desired speed in meters/second.

`PEAKWIDTH:` The width of the peak in meters/second in the produced objective function.

`BASEWIDTH:` The width of the base, in meters/second in the produced objective function.

`DURATION:` This is a 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.

11.9 BHV_GoToDepth

This behavior will drive the vehicle to a sequence of specified depths and duration at each depth. 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`. 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:

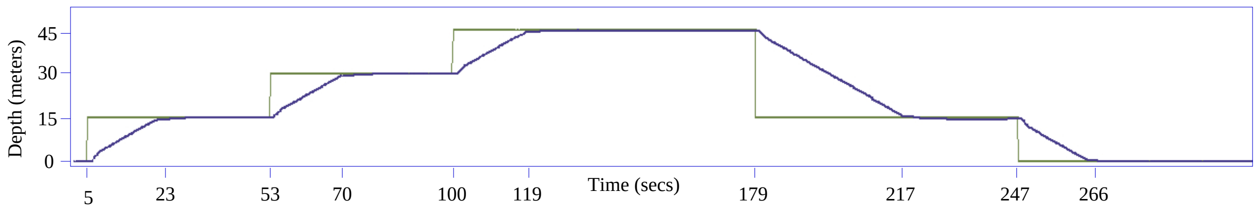


Figure 44: Depth log from simulation with the depth parameters shown in Listing 8. The lighter, step-like line indicates the values of `DESIRED_DEPTH` generated by the helm, and the darker line indicates the recorded depth value of the vehicle. The depth plateaus start from the moment the vehicle achieves 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 8.

`DEPTH:` 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.

`REPEAT:` 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.

`PERPETUAL:` If equal to *true*, when the vehicle completes its evolution of depths (perhaps several evolutions if `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*.

`CAPTURE_DELTA:` 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.

`CAPTURE_FLAG:` The name of a MOOS variable incremented each time a target depth level has been achieved. Useful for logfile debugging/analyzing 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.

11.10 BHV_MemoryTurnLimit

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:

`MEMORY_TIME`: The duration of time for which the heading history is maintained and heading average calculated.

`TURN_RANGE`: The range of heading values deviating from the current heading average outside of which the behavior reflects sharp penalty in its objective function.

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 $[0, 359]$.

$$\text{heading\_avg} = \text{atan2}(s, c) \cdot 180/\pi,$$

where s and c are given by:

$$s = \sum_{k=0}^{n-1} \sin(h_k \pi/180)), \qquad c = \sum_{k=0}^{n-1} \cos(h_k \pi/180)).$$

The vehicle turn radius r is not explicitly a parameter of the behavior, but is given by:

$$r = v/((u/180)\pi),$$

where v is the vehicle speed and u is the turn rate given by:

$$u = \texttt{TURN\_RANGE}/\texttt{MEMORY\_TIME}.$$

The same turn radius is possible with different pairs of values for `TURN_RANGE` and `MEMORY_TIME`. However,m larger values of `TURN_RANGE` allow sharper initial turns but temper the turn rate after the initial sharper turn has been achieved.

A Rendering of the MemoryTurnLimit Objective Function

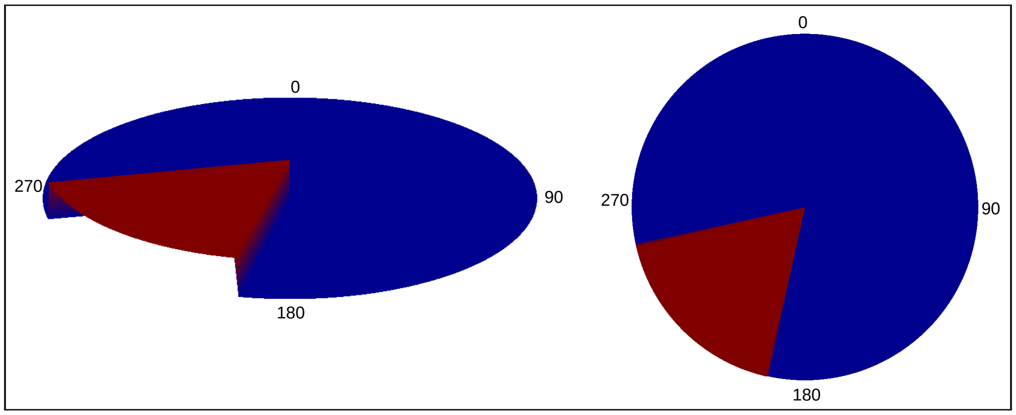


Figure 45: **The MemoryTurnLimit objective function**: The objective function produced by the MemoryTurn-Limit behavior is defined over possible heading values. Depicted here is an objective function formed when the recent heading history is 225 degrees and the `turn_range` parameter is set to 30 degrees. 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.

11.11 BHV_StationKeep

11.11.1 Overview of the BHV_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.

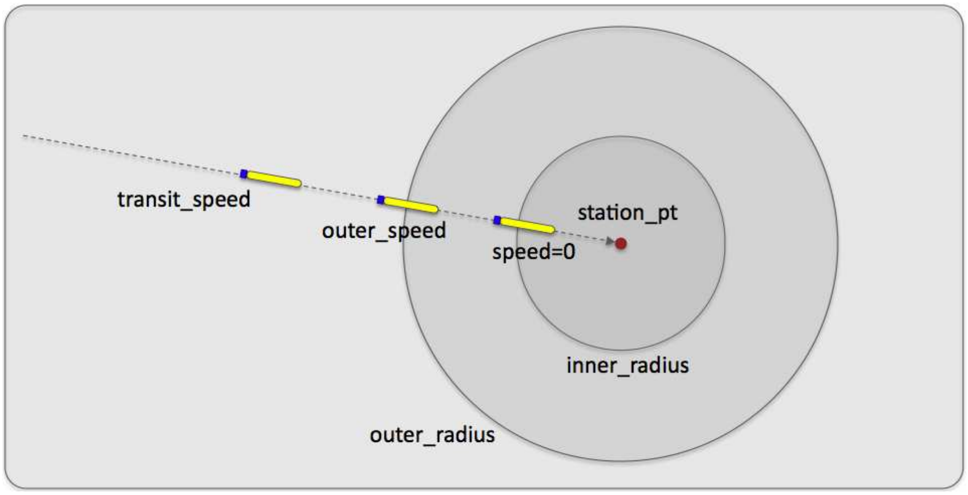


Figure 46: **The station-keep behavior parameters:** The station-keep behavior can be configured to approach the outer station circle with a given transit speed, and will decrease its preference for speed linearly between the outer radius and inner radius. The preferred speed is zero when the vehicle 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 `BHV_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.

11.11.2 Brief Summary of the BHV_StationKeep Behavior Parameters

The following parameters are defined for this behavior. A more detailed description is provided other parts of this section, and in Table 21 on page 157.

`STATION_PT`:	An x,y pair given as a point in local coordinates.
`POINT`:	A supported alias for `STATION_POINT`.
`CENTER_ACTIVE`:	If `true`, station-keep at position upon activation.
`INNER_RADIUS`:	Distance to station-point within which the preferred speed is zero.
`OUTER_RADIUS`:	Distance within which the preferred speed begins to decrease.
`OUTER_SPEED`:	Preferred speed at outer radius, decreasing toward inner radius.
`SWING_TIME`:	Duration of drift of station circle with vehicle upon activation.
`TRANSIT_SPEED`:	Preferred speed beyond the outer radius.
`EXTRA_SPEED`:	A deprecated alias for `TRANSIT_SPEED`.
`PASSIVE_STATION_RADIUS`:	A radius used for low-power, passive station-keeping.
`PASSIVE_STATION_VARIABLE`:	Name of MOOS variable used for conveying passive-station mode.

11.11.3 Setting the Station-Keep Point and Radial-Speed Relationships

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_active = true  // "true" is case insensitive
```

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:

```
swing_time = <time-duration>   // default is 0
```

The `<time-duration>` is given in seconds and the duration is clipped by the range $[0, 60]$.

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. (See Section 6.5.3 for more detail on behavior run states.) In this situation, a warning will be posted: `BHV_WARNING="STATION_POINT_NOT_SET"`.

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 then 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.

11.11.4 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 `PASSIVE_STATION_RADIUS` before it resumes outputting a preferred heading and non-zero speed. The idea is shown in Figure 47.

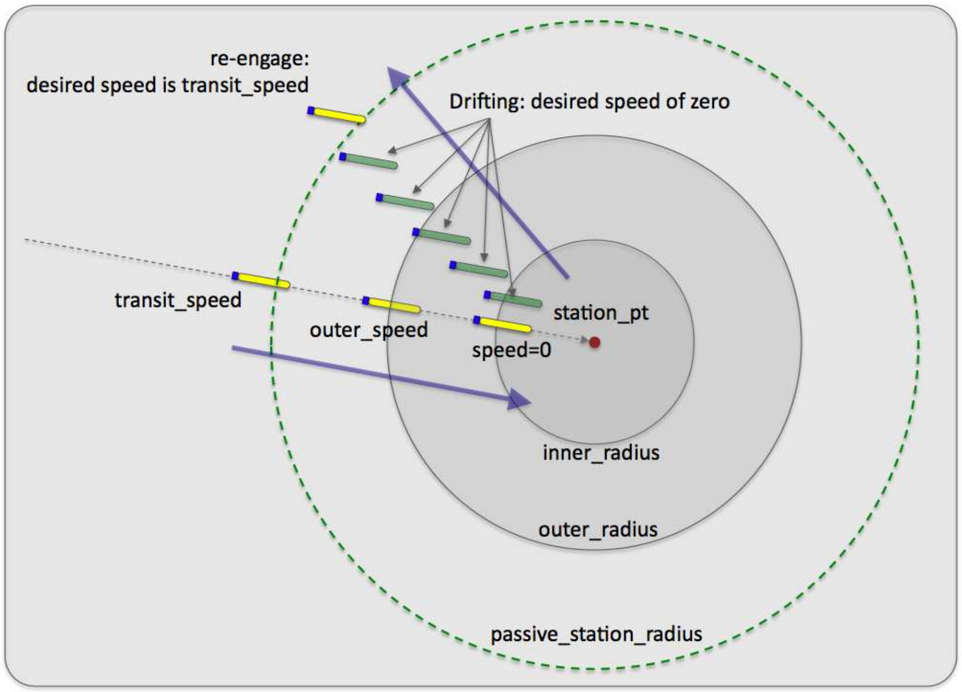


Figure 47: **Passive station-keeping:** The station-keep behavior can be configured in the "passive" mode. The vehicle will move toward the station point until it reaches the inner_radius or until progress ceases. It will then drift until its distance to the station point is beyond the passive_station_radius. 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 48.

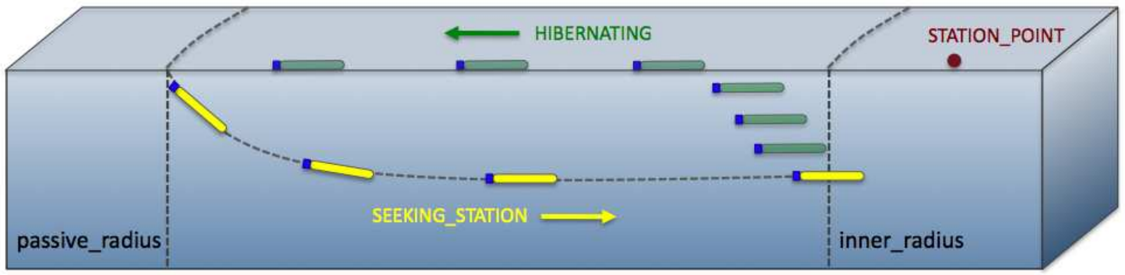


Figure 48: **Passive station-keeping with depth coordination:** The passive mode can be coordinated with the ConstantDepth behavior to dive each time the StationKeep behavior enters the `"SEEKING_STATION"` mode. This ensures that a UUV needing to be at depth to have reliable heading control will indeed be at depth when it needs to be.

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 `"STATION_SEEKING"` 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 `"STATION_SEEKING"` 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 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.

11.11.5 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_ACTIVE = 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 7.2.2 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 station-keep 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`, `uTermCommand`, `pMarineViewer` or any third party command and control interface.

11.12 BHV_Timer

This behavior can nearly be considered a no-op behavior; it has 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`, `CONDITION`, `RUNFLAG` and `ENDFLAG` parameters are all defined generally for behaviors. There are no additional parameters defined for this behavior.

12 Multi-Vehicle Behaviors of the IvP Helm

The following is a description of some behaviors currently written for the IvP Helm that reason about relative position to another vehicle. Each such behavior needs to know about the position of a given contact. Currently we simply assume that a contact's ID or vehicle name is known a priori and its position information arrives in the MOOSDB in the form of an AIS report (discussed earlier, and again in the section on example scenarios). Currently work is addressing the development of a separate MOOS process acting as a contact manager and perhaps spawning behaviors dynamically. At this point however, behaviors relating to the relative position of another vehicle are configured statically.

12.1 Parameters Common All Multi-Vehicle Behaviors

The following set of parameters are common to all the multi-vehicle behaviors described in later sections.

`CONTACT:` The name of the contact.

`ON_NO_CONTACT_OK:` The name of the contact.

`EXTRAPOLATE:` Boolean controlling whether the contact position is extrapolated from the last known position using the associated speed and heading. This feature is particularly important when position updates are sparse, e.g. for underwater vehicles using acoustic communication. The time delays for which extrapolation will be applied are controlled by the `DECAY` parameters.

`DECAY:` This parameter takes two arguments separated by a comma. The first argument is the decay start time (in seconds), and the second is the decay end time (also in seconds). The behavior extrapolates the contact position based on the last known position, heading and speed. The speed of the contact begins to *decay* based on the time since the last contact update. This is a safeguard against perpetually trailing a vehicle the ceases to provide a contact report. The default is 5 and 10 seconds respectively.

12.2 BHV_AvoidCollision

This behavior will drive the vehicle to avoid collisions with another specified vehicle. It reasons over the "closest point of approach" (CPA) of candidate ownship actions. The following parameters are defined for this behavior:

`ACTIVE_OUTER_DISTANCE:` The distance (meters) to the specified other vehicle, below which the behavior will begin to be relevant (have a non-zero priority weight). At higher distances, the behavior will not contribute an objective function.

`ACTIVE_INNER_DISTANCE:` The distance (meters) to the specified other vehicle, at which the behavior will apply 100% of its assigned priority weight. Ranges smaller than this distance will also have full priority weight.

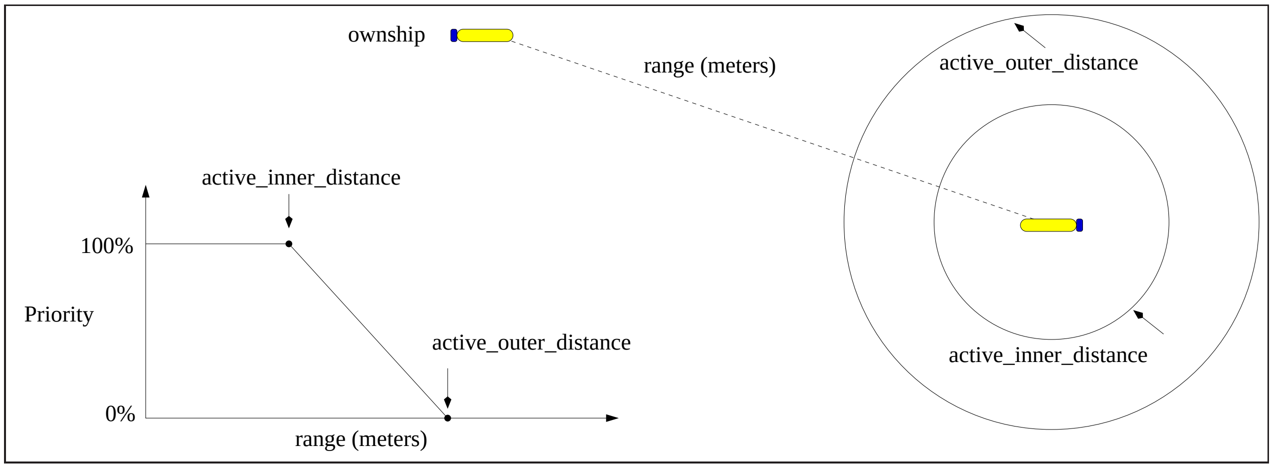


Figure 49: **Parameters for the BHV_AvoidCollision behavior:** The *ownship* vehicle is the platform running the helm. The range between the two vehicles affects whether the behavior is active and with what priority weight. Beyond the `active_outer_distance`, the behavior is not active. Within the `active_inner_distance`, the behavior is active with 100% of its priority weight.

`COLLISION_DISTANCE:` The distance (in meters) between ownship and the contact at the closest point of approach (CPA) for a candidate maneuver, below which the behavior treats the distance as it would an actual collision between the two vehicles.

`ALL_CLEAR_DISTANCE:` The distance (in meters) between ownship and the contact at the closest point of approach (CPA) for a candidate maneuver, above which the behavior treats the distance as having the maximum utility.

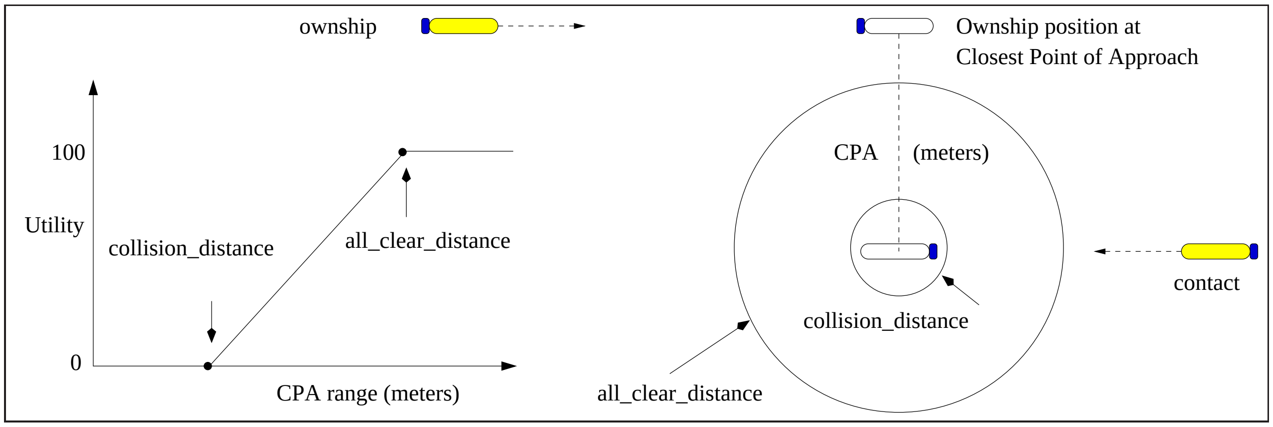


Figure 50: Parameters for the BHV_AvoidCollision behavior. The *ownship* vehicle is the platform running the helm. The `collision_distance` is used when applying a utility metric to a calculated closest point of approach (CPA) for a candidate maneuver. A CPA less than or equal to the `collision_distance` is treated as an actual collision with the lowest utility rating.

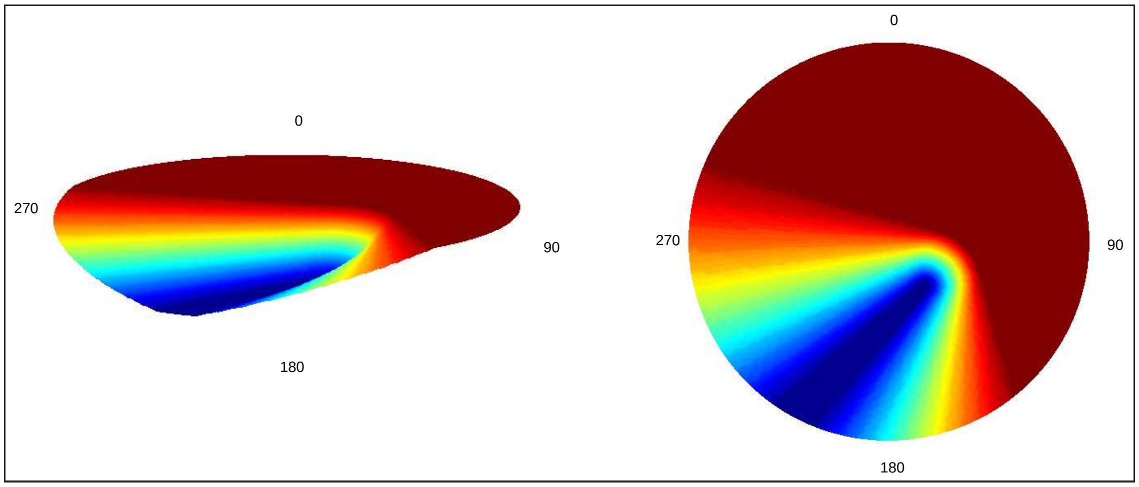


Figure 51: The objective function produced by the collision avoidance behavior is defined over possible heading and speed values. Higher speeds are represented farther radially out from the center.

A Rendering of the Collision Avoidance Function

12.3 BHV_CutRange

This behavior will drive the vehicle to reduce the range between itself and another specified vehicle (nearly the opposite of the BHV_AvoidCollision behavior). The following parameters are defined for this behavior:

`DIST_PRIORITY_INTERVAL:` Two distance values given by a comma-separated pair `min,max` where the min value is the range at or below which the behavior will have a zero priority. The min value is the range at or above which the behavior will have 100% of its statically assigned priority. The percentage between the two values scales linearly.

`TIME_ON_LEG:` The behavior uses a closest-point-of-approach (CPA) calculation to evaluate candidate heading-speed maneuvers. The CPA calculation is based on a 60 second maneuver by default, but this time duration can be altered with this parameter.

`GIVE_UP_RANGE:` The range between ownship and the contact at or above which the behavior will cease to provide output (the objective function) to influence the vehicle heading and speed. By default this value is zero which is interpreted as infinity - it will never give up.

`PATIENCE:` The `PATIENCE` parameter ranges between `0` and `100` and is clipped automatically if out of range. A value of `0` will result in the behavior attempting to steer the vehicle directly toward the current position of the contact. A value of 100 will result in an attempt to steer toward the closest point of approach given the current linear track of the contact, and the prevailing setting of the `TIME_ON_LEG` parameter.

12.4 BHV_Shadow

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. The following parameters are defined for this behavior:

`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.

`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.

12.5 BHV_Trail

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:

`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.

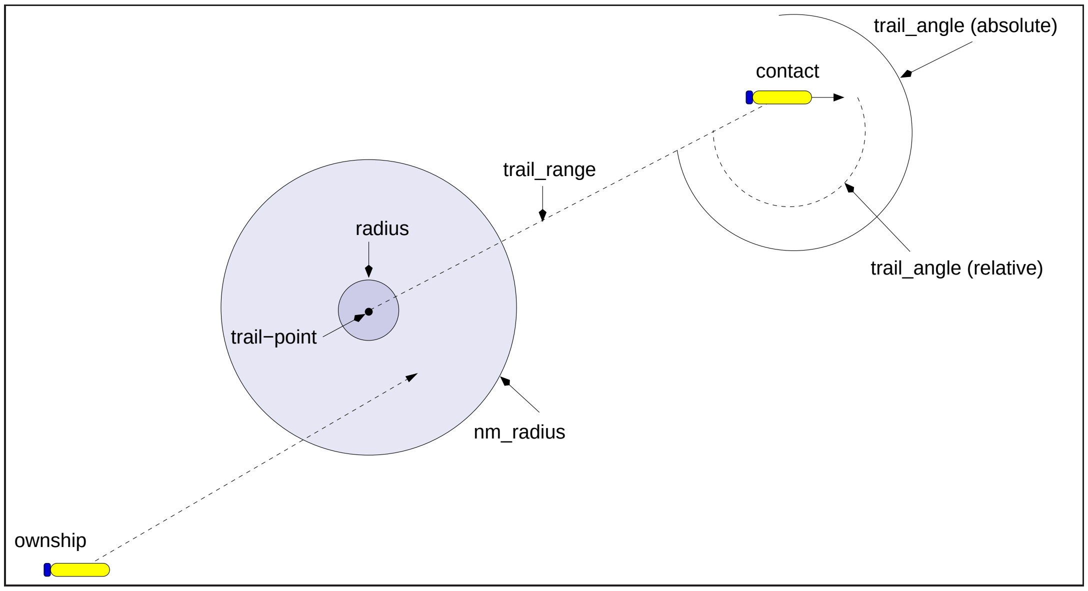


Figure 52: Interpolation of vehicle speed inside the radius set by `NM_RADIUS` relative to the extrapolated trail position.

.

`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 Fig. 52. 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.

13 Appendix - Behavior Summaries

Parameter Summary for BHV_Waypoint

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
points, polygon	string	`0,0:45,0:45,80:0,0`	yes	-	118
capture_radius	double	`7`	-	`0`	119
lead	double	`10`	no	`-1`	119
nm_radius	double	`18`	no	`0`	119
order	string	`reverse`	no	`normal`	118
post_suffix	string	`IKE`	yes	`""`	118
repeat	int	`3`	no	`0`	118
speed	double	`1.2`	-	`0`	118
wpt_status_var	`MOOSVAR`	`WPT_REPORT`	yes	`WPT_STAT`	121
wpt_index_var	`MOOSVAR`	`WPT_IX`	yes	`WPT_INDEX`	121
cycle_index_var	`MOOSVAR`	`CYCLE_IX`	yes	`CYCLE_INDEX`	121
cycleflag	`MOOSVAR=value`	`CYCLED=true`	yes	-	121

Table 11: Parameters for the BHV_Waypoint behavior.

Example Behavior File Configuration for BHV_Waypoint

Listing 13.1 - 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

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
polygon	string	`0,0:45,0:45,80:0,80:0,0`	-	-	122
max_depth	double	`200`	-	`0`	123
min_altitude	double	`25`	-	`0`	123
max_time	double	`3600`	-	`0`	123
trigger_entry_time	double	`1.5`	-	`0`	122
trigger_exit_time	double	`2.4`	-	`0`	122

Table 12: Parameters for the BHV_OpRegion behavior.

Example Behavior File Configuration for BHV_OpRegion

Listing 13.2 - An example BHV_OpRegion configuration.

```
Behavior = BHV_OpRegion
{
  name               = bhv_opregion
  polygon            = label,opregion : -57,-60 : -70,-109 : -77,-144

  max_depth          = 50     // meters
  min_altitude       = 10     // meters
  max_time           = 3600   // seconds
  trigger_entry_time = 0.5    // seconds
  trigger_exit_time  = 1.0    // seconds
}
```

Parameter Summary for BHV_Loiter

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
polygon	string	`0,0:45,0:45,80:0,80:0,0`	yes	-	125
speed	double	`1.5`	-	`0`	125
radius	double	`10`	-	`0`	125
nm_radius	double	`25`	-	`0`	125
clockwise	string	`FALSE`	no	`true`	125
acquire_dist	double	`15`	-	`10`	125
post_suffix	string	`REGION-1`	yes	`""`	125

Table 13: Parameters for the BHV_Loiter behavior.

Example Behavior File Configuration for BHV_Loiter

Listing 13.3 - An example BHV_Loiter configuration.

```
Behavior = BHV_Loiter
{
  name     = loiter_alpha
  pwt      = 100
  duration = 3600 // One hour
  updates  = LOITER_ALPHA_UPDATES

      polygon = radial:100,-100,80,12
        speed = 3.0
       radius = 8.0
    nm_radius = 16.0
    clockwise = true
  aquire_dist = 25
}
```

Parameter Summary for BHV_PeriodicSpeed

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
period_length	double	`60`	-	`0`	127
period_gap	double	`600`	-	`0`	127
period_speed	double	`0.8`	-	`0`	127
period_peakwidth	double	`0.2`	-	`0`	127
period_basewidth	double	`0.5`	-	`0`	127
stat_pending_inactive	MOOSVAR	`PS_PENDING_INACTIVE`	yes	-	127
stat_pending_active	MOOSVAR	`PS_PENDING_ACTIVE`	yes	-	127

Table 14: Parameters for the BHV_PeriodicSpeed behavior.

Example Behavior File Configuration for BHV_PeriodicSpeed

Listing 13.3 - An example BHV_PeriodicSpeed configuration.

```
Behavior = BHV_PeriodicSpeed
{
  name     = periodic_speed
  priority = 500

     period_length      = 30       // seconds
        period_gap      = 120      // seconds
      period_speed      = 0.5      // meters/sec
  period_peakwidth      = 0.1
  period_basewidth      = 0.5
  stat_pending_active   = PS_PENDING_ACTIVE
  stat_pending_inactive = PS_PENDING_INACTIVE
}
```

Parameter Summary for BHV_PeriodicSurface

Parameter	Argument Type	Example	Case-Sense	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
period	double	`60`	-	`300`	128
mark_variable	MOOSVAR	`GPS_RECEIVED`	yes	`GPS_UPDATE_RECEIVED`	129
atsurface_status_variable	MOOSVAR	`TIME_AT_SURFACE`	yes	`TIME_AT_SURFACE`	129
pending_status_variable	MOOSVAR	`PENDING_SURFACE`	yes	`PENDING_SURFACE`	129
ascent_speed	double	`1.0`	-	`*`	129
ascent_grade	string	`quasi`	no	`linear`	129
zero_speed_depth	double	`2.5`	-	`0`	129
max_time_at_surface	MOOSVAR	`60`	yes	`300`	129

Table 15: Parameters for the BHV_PeriodicSurface behavior.

Example Behavior File Configuration for BHV_PeriodicSurface

Listing 13.4 - An example BHV_PeriodicSurface configuration.

```
Behavior = BHV_PeriodicSurface
{
  name          = bhv_periodic_surface
  priority      = 500
  active_flag   = SURFACING, IN_PROGRESS
  inactive_flag = SURFACING, NO

              period = 3600     // seconds
        ascent_speed = 1.0      // meters per second
    zero_speed_depth = 2.5      // meters
  max_time_at_surface = 120     // seconds
        ascent_grade = linear
       mark_variable = GPS_UPDATE_RECEIVED
     status_variable = PERIODIC_PENDING_SURFACE
}
```

Parameter Summary for BHV_ConstantDepth

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
depth	double	`35`	-	`0`	130
peakwidth	double	`5`	-	`0`	130
basewidth	double	`15`	-	`2`	130

Table 16: Parameters for the BHV_ConstantDepth behavior.

Example Behavior File Configuration for BHV_ConstantDepth

Listing 13.5 - An example BHV_ConstantDepth configuration.

```
Behavior = BHV_ConstantDepth
{
  // General Behavior Parameters
  name      = constant_depth_survey
  priority  = 100
  condition = AUTONOMY_MODE = SURVEY
  duration  = no-time-limit
  updates   = NEW_SURVEY_DEPTH
  nostarve  = NAV_DEPTH, 3.0

  // BHV_ConstantDepth Behavior Parameters
      depth = 50      // meters
  peakwidth = 5
  basewidth = 10
}
```

Parameter Summary for BHV_ConstantHeading

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
heading	double	`35`	-	`0`	130
peakwidth	double	`5`	-	`10`	130
basewidth	double	`175`	-	`170`	130

Table 17: Parameters for the BHV_ConstantHeading behavior.

Example Behavior File Configuration for BHV_ConstantHeading

Listing 13.6 - An example BHV_ConstantHeading configuration.

```
Behavior = BHV_ConstantHeading
{
  name      = bhv_constant_heading
  priority  = 100
  duration  = 60
  condition = AUTONOMY_MODE = PID_TEST
  updates   = NEW_TEST_HEADING
  nostarve  = NAV_HEADING, 3.0

    heading = 45  // degrees
  peakwidth =  0
  basewidth =  5
}
```

Parameter Summary for BHV_ConstantSpeed

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
speed	double	`1.2`	-	`0.0`	131
peakwidth	double	`0.1`	-	`0.0`	131
basewidth	double	`0.6`	-	`2.0`	131

Table 18: Parameters for the BHV_ConstantSpeed behavior.

Example Behavior File Configuration for BHV_ConstantSpeed

Listing 13.7 - An example BHV_ConstantSpeed configuration.

```
Behavior = BHV_ConstantSpeed
{
  name        = const_speed_bravo
  priority    = 100
  duration    = 60
  active_flag = BRAVO_SPEED_TEST = in-progress
  nostarve    = NAV_SPEED, 2.0

  speed     = 1.8  // meters per second
  peakwidth = 0.3
  basewidth = 1.0
}
```

Parameter Summary for BHV_GoToDepth

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
depth, depths	string	`50,10:40,60`	yes	-	132
repeat	int	`5`	-	`0`	132
capture_delta	double	`2`	-	`2.5`	132
capture_flag	MOOSVAR	`DEPTH_HIT`	yes	-	132

Table 19: Parameters for the BHV_GoToDepth behavior.

Example Behavior File Configuration for BHV_GoToDepth

Listing 13.8 - An example BHV_GoToDepth configuration.

```
Behavior = BHV_GoToDepth
{
  name      = goto_depth_set_alpha
  priority  = 100
  condition = DEPLOY == true
  endflag   = GOTO_DEPTH_ALPHA = DONE

         depths = 15,30: 30,30: 45,60: 15,30
  capture_delta = 1  // meters
   capture_flag = DEPTH_LEVELS_ACHIEVED
}
```

Parameter Summary for BHV_MemoryTurnLimit

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
memory_time	double	`60`	-	`-1`	134
turn_range	double	`45`	-	`-1`	134

Table 20: Parameters for the BHV_MemoryTurnLimit behavior.

Example Behavior File Configuration for BHV_MemoryTurnLimit

Listing 13.9 - An example BHV_MemoryTurnLimit configuration.

```
zv
Behavior = BHV_MemoryTurnLimit
{
  name      = memturnlimit
  priority  = 1000

  memory_time = 60  // seconds
   turn_range = 35  // degrees
}
```

Parameter Summary for BHV_StationKeep

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
station_pt, point	string	`50,75`	yes	`0,0`	137
center_activate	string	`TRUE`	no	`false`	137
inner_radius	double	`10`	-	`4`	137
outer_radius	double	`25`	-	`15`	137
outer_speed	double	`1.8`	-	`1.2`	137
transit_speed	double	`1.9`	-	`2.5`	137
passive_station_radius	double	`200`	-	`0`	138
passive_station_variable	MOOSVAR	`PSK_MODE_CHARLIE`	-	`PSKEEP_MODE`	138

Table 21: Parameters for the BHV_StationKeep behavior.

Example Behavior File Configuration for BHV_StationKeep

Listing 13.10 - An example BHV_StationKeep configuration.

```
Behavior = BHV_StationKeep
{
  name       = bhv_station_keep
  priority   = 100
  condition  = (ON_STATION=true) and (RETURN=false)
  updates    = STATION_UPDATES

                station_pt = 200,-150
           center_activate = true
              inner_radius = 10
              outer_radius = 40
               outer_speed = 0.8
             transit_speed = 1.8
    passive_station_radius = 400          // meters
  passive_station_variable = PSKEEP_MODE  // the default
}
```

Parameter Summary for BHV_Timer

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
No additional parameters for this behavior					

Table 22: Parameters for the BHV_Timer behavior.

Example Behavior File Configuration for BHV_Timer

Listing 13.11 - An example BHV_Timer configuration.

```
Behavior = BHV_Timer
{
  name      = bhv_timer_a
  duration  = 60            // seconds
  condition = loiter = alpha
  end_flag  = loiter = beta
}
```

Parameter Summary for BHV_AvoidCollision

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
contact	string	Alliance	yes	-	142
on_no_contact_ok	boolean	true	no	`true`	142
extrapolate	boolean	true	no	`false`	142
decay	double,double	10, 30	-	`0,0`	142
active_inner_distance	double	50	-	`50`	142
active_outer_distance	double	200	-	`200`	142
all_clear_distance	double	100	-	`75`	143
collision_distance	double	10	-	`10`	143

Table 23: Parameters for the BHV_AvoidCollision behavior.

Example Behavior File Configuration for BHV_AvoidCollision

Listing 13.12 - An example BHV_AvoidCollision configuration.

```
Behavior = BHV_AvoidCollision
{
  name      = avoid_collision_alpha
  pwt       = 100
  condition = AVOIDANCE_MODE != INACTIVE

                contact = alpha
   active_outer_distance = 150
   active_inner_distance = 75
      collision_distance = 15
      all_clear_distance = 80
            active_grade = linear
        on_no_contact_ok = true
             extrapolate = true
                   decay = 30,60
}
```

Parameter Summary for BHV_CutRange

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
contact	string	Alliance	yes	-	142
on_no_contact_ok	boolean	true	no	`true`	142
extrapolate	boolean	true	no	`false`	142
decay	double,double	10, 30	-	`0,0`	142
dist_priority_interval	double,double	`40,100`	-	`0,0`	144
time_on_leg	double	`60`	-	`15`	144
give_up_range	double	`500`	-	`0`	144
patience	double	`50`	-	`0`	144

Table 24: Parameters for the BHV_CutRange behavior.

Example Behavior File Configuration for BHV_CutRange

Listing 13.13 - An example BHV_CutRange configuration.

```
Behavior = BHV_CutRange
{
  name         = bhv_cutrange
  pwt          = 100
  contact      = zulu

  dist_priority_interval = 25,100
             time_on_leg = 60
           give_up_range = 400
                patience = 75
}
```

Parameter Summary for BHV_Shadow

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
contact	string	Alliance	yes	-	142
on_no_contact_ok	boolean	true	no	`true`	142
extrapolate	boolean	true	no	`false`	142
decay	double,double	10, 30	-	`0,0`	142
max_range	double	`100`	-	`0`	144
heading_peakwidth	double	`10`	-	`20`	144
heading_basewidth	double	`170`	-	`160`	144
speed_peakwidth	double	`0.3`	-	`0.1`	144
speed_basewidth	double	`0.5`	-	`2.0`	144

Table 25: Parameters for the BHV_Shadow behavior.

Example Behavior File Configuration for BHV_Shadow

Listing 13.14 - An example BHV_Shadow configuration.

```
Behavior = BHV_Shadow
{
  name     = bhv_shadow
  pwt      = 100
  contact  = delta

          max_range = 200
  heading_peakwidth = 10
  heading_basewidth = 170
    speed_peakwidth = 10
    speed_basewidth = 170
}
```

Parameter Summary for BHV_Trail

Parameter	Argument Type	Example	Case-Sensitive	Default	Page
name	string	`loiter-west-zone`	yes	*mandatory*	71
duration	double	`600`	-	`-1`	74
duration_status	MOOSVAR	`loiter_remaining`	yes	-	74
priority, pwt	double	`100`	-	`100`	71
runflag	MOOSVAR=value	`LOITERING = maybe`	yes	-	62
endflag	MOOSVAR=value	`LOITERING = done`	yes	-	62
activeflag	MOOSVAR=value	`LOITERING = yes`	yes	-	62
inactiveflag	MOOSVAR=value	`LOITERING = off`	yes	-	62
idleflag	MOOSVAR=value	`LOITERING = no`	yes	-	62
nostarve	MOOSVAR,double	`INFO,60`	yes	-	75
perpetual	string	`false`	no	`false`	75
updates	MOOSVAR	`LOITER_INFO`	yes	-	73
condition	Logic Expression	`QUALITY <= 7`	yes	-	61
contact	string	Alliance	yes	-	142
on_no_contact_ok	boolean	true	no	`true`	142
extrapolate	boolean	true	no	`false`	142
decay	double,double	10, 30	-	`0,0`	142
trail_range	double	`20`	-	`50`	144
trail_angle	double	`270`	-	`180`	144
trail_angle_type	string	`absolute`	no	`relative`	144
radius	double	`8`	-	`5`	144
nm_radius	double	`20`	-	`20`	144
max_range	double	`50`	-	`0`	144

Table 26: Parameters for the BHV_Trail behavior.

Example Behavior File Configuration for BHV_Trail

Listing 13.15 - An example BHV_Trail configuration.

```
Behavior = BHV_Trail
{
  name             = bhv_trail
  priority         = 100

  contact          = delta
  extrapolate      = true
  on_no_contact_ok = true
  decay            = 20,60    // seconds

       trail_range = 50       // meters
       trail_angle = 185      // degrees
  trail_angle_type = relative
            radius = 10       // meters
         nm_radius = 30       // meters
         max_range = 300      // meters
}
```

14 Appendix - 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".

antiquewhite, (fa,eb,d7)
aqua (00,ff,ff)
aquamarine (7f,ff,d4)
azure (f0,ff,ff)
beige (f5,f5,dc)
bisque (ff,e4,c4)
black (00,00,00)
blanchedalmond(ff,eb,cd)
blue (00,00,ff)
blueviolet (8a,2b,e2)
brown (a5,2a,2a)
burlywood (de,b8,87)
cadetblue (5f,9e,a0)
chartreuse (7f,ff,00)
chocolate (d2,69,1e)
coral (ff,7f,50)
cornsilk (ff,f8,dc)
cornflowerblue(64,95,ed)
crimson (de,14,3c)
cyan (00,ff,ff)
darkblue (00,00,8b)
darkcyan (00,8b,8b)
darkgoldenrod (b8,86,0b)
darkgray (a9,a9,a9)
darkgreen (00,64,00)
darkkhaki (bd,b7,6b)
darkmagenta (8b,00,8b)
darkolivegreen(55,6b,2f)
darkorange (ff,8c,00)
darkorchid (99,32,cc)
darkred (8b,00,00)
darksalmon (e9,96,7a)
darkseagreen (8f,bc,8f)
darkslateblue (48,3d,8b)
darkslategray (2f,4f,4f)
darkturquoise (00,ce,d1)
darkviolet (94,00,d3)
deeppink (ff,14,93)
deepskyblue (00,bf,ff)
dimgray (69,69,69)
dodgerblue (1e,90,ff)
firenrick (b2,22,22)
floralwhite (ff,fa,f0)
forestgreen (22,8b,22)
fuchsia (ff,00,ff)
gainsboro (dc,dc,dc)
ghostwhite (f8,f8,ff)
gold (ff,d7,00)
goldenrod (da,a5,20)
gray (80,80,80)
green (00,80,00)
greenyellow (ad,ff,2f)
honeydew (f0,ff,f0)
hotpink (ff,69,b4)
indianred (cd,5c,5c)
indigo (4b,00,82)
ivory (ff,ff,f0)
khaki (f0,e6,8c)
lavender (e6,e6,fa)
lavenderblush (ff,f0,f5)
lawngreen (7c,fc,00)
lemonchiffon (ff,fa,cd)
lightblue (ad,d8,e6)
lightcoral (f0,80,80)
lightcyan (e0,ff,ff)
lightgoldenrod(fa,fa,d2)
lightgray (d3,d3,d3)
lightgreen (90,ee,90)
lightpink (ff,b6,c1)
lightsalmon (ff,a0,7a)
lightseagreen (20,b2,aa)
lightskyblue (87,ce,fa)
lightslategray(77,88,99)
lightsteelblue(b0,c4,de)

lightyellow (ff,ff,e0)
lime (00,ff,00)
limegreen (32,cd,32)
linen (fa,f0,e6)
magenta (ff,00,ff)
maroon (80,00,00)
mediumblue (00,00,cd)
mediumorchid (ba,55,d3)
mediumseagreen(3c,b3,71)
mediumslateblue(7b,68,ee)
mediumspringgreen(00,fa,9a)
mediumturquoise(48,d1,cc)
mediumvioletred(c7,15,85)
midnightblue (19,19,70)
mintcream (f5,ff,fa)
mistyrose (ff,e4,e1)
moccasin (ff,e4,b5)
navajowhite (ff,de,ad)
navy (00,00,80)
oldlace (fd,f5,e6)
olive (80,80,00)
olivedrab (6b,8e,23)
orange (ff,a5,00)
orangered (ff,45,00)
orchid (da,70,d6)
palegreen (98,fb,98)
paleturquoise (af,ee,ee)
palevioletred (db,70,93)
papayawhip (ff,ef,d5)
peachpuff (ff,da,b9)
pelegoldenrod (ee,e8,aa)
peru (cd,85,3f)
pink (ff,c0,cb)
plum (dd,a0,dd)
powderblue (b0,e0,e6)
purple (80,00,80)
red (ff,00,00)
rosybrown (bc,8f,8f)
royalblue (41,69,e1)
saddlebrowm (8b,45,13)
salmon (fa,80,72)
sandybrown (f4,a4,60)
seagreen (2e,8b,57)
seashell (ff,f5,ee)
sienna (a0,52,2d)
silver (c0,c0,c0)
skyblue (87,ce,eb)
slateblue (6a,5a,cd)
slategray (70,80,90)
snow (ff,fa,fa)
springgreen (00,ff,7f)
steelblue (46,82,b4)
tan (d2,b4,8c)
teal (00,80,80)
thistle (d8,bf,d8)
tomatao (ff,63,47)
turquoise (40,e0,d0)
violet (ee,82,ee)
wheat (f5,de,b3)
white (ff,ff,ff)
whitesmoke (f5,f5,f5)
yellow (ff,ff,00)
yellowgreen (9a,cd,32)

References

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

[2] Ronald C. Arkin, William M. Carter, and Douglas C. Mackenzie. Active Avoidance: Escape and Dodging Behaviors for Reactive Control. *International Journal of Pattern Recognition and Artificial Intelligence*, 5(1):175–192, 1993.

[3] Michael R. Benjamin. The Interval Programming Model for Multi-Objective Decision Making. Technical Report AIM-2004-021, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, September 2004.

[4] Michael R. Benjamin. MOOS-IvP Autonomy Tools Users Manual. Technical Report MIT-CSAIL-TR-2008-065, MIT Computer Science and Artificial Intelligence Lab, November 2008.

[5] Michael R. Benjamin and Joe Curcio. COLREGS-Based Navigation in Unmanned Marine Vehicles. In *AUV-2004*, Sebasco Harbor, Maine, June 2004.

[6] Michael R. Benjamin, Paul M. Newman, Henrik Schmidt, and John J. Leonard. A Tour of MOOS-IvP Autonomy Software Modules. Technical Report MIT-CSAIL-TR-2009-006, MIT Computer Science and Artificial Intelligence Lab, January 2009.

[7] Mike Benjamin, Henrik Schmidt, and John J. Leonard. `http://www.moos-ivp.org`.

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

[9] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, April 1986.

[10] Marc Carreras, J. Batlle, and Pere Ridao. Reactive Control of an AUV Using Motor Schemas. In *International Conference on Quality Control, Automation and Robotics*, Cluj Napoca, Rumania, May 2000.

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

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

[13] Ratnesh Kumar and James A. Stover. A Behavior-Based Intelligent Control Architecture with Application to Coordination of Multiple Underwater Vehicles. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Cybernetics*, 30(6):767–784, November 2001.

[14] Paul Newman. `http://www.robots.ox.ac.uk/~pnewman/TheMOOS/`.

[15] Paul M. Newman. MOOS - A Mission Oriented Operating Suite. Technical Report OE2003-07, MIT Department of Ocean Engineering, 2003.

[16] Paolo Pirjanian. *Multiple Objective Action Selection and Behavior Fusion*. PhD thesis, Aalborg University, 1998.

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

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

[19] Julio K. Rosenblatt, Stefan B. Williams, and Hugh Durrant-Whyte. Behavior-Based Control for Autonomous Underwater Exploration. *International Journal of Information Sciences*, 145(1-2):69–87, 2002.

[20] Stefan B. Williams, Paul Newman, Gamini Dissanayake, Julio K. Rosenblatt, and Hugh Durrant-Whyte. A decoupled, distributed `AUV` control architecture. In *Proceedings of 31st International Symposium on Robotics*, pages 246–251, Montreal, Canada, 2000.

Index