Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-006 February 13, 2009

# A Tour of MOOS-IvP Autonomy Software Modules

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

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# A Tour of MOOS-IvP Autonomy Software Modules

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

January 6th, 2009 - Release 3.1 (SVN Revision 1668)

**Abstract**

This paper provides an overview of the MOOS-IvP autonomy software modules. The MOOS-IvP collection of software, i.e., codebase, described here has been developed and is currently maintained by three organizations - Oxford University, Massachusetts Institute of Technology (MIT), and the Naval Undersea Warfare Center (NUWC) Division Newport Rhode Island. The objective of this paper is to provide a comprehensive list of modules and provide for each (a) a general description of functionality, (b) dependency relationships to other modules, (c) rough order of magnitude in complexity or size, (d) authorship, and (e) current and planned distribution access.

| | |
|---|---|
| MOOS Applications: | 23 |
| IvP Helm Behaviors: | 17 |
| Non-MOOS Utility Applications: | 7 |
| Unique lines of code: | 101,154 |
| Aggregate lines of code: | 569,340 |

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

**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 a catalog style overview of modules written under the umbrella term MOOS-IvP. The scope of discussion includes, for each module, a brief description of the module function, authorship, source for download, rough measure of complexity, and module dependencies. By the term *module* we mean a group of C++ code that compiles into either an actual runnable application, or compiles into a library for linking into one or more applications. The term MOOS-IvP refers to two distinct architectures, discussed further in the next section. Each architectures supports a plug-and-play style of development through the incremental development of modules, each adding a distinct function. With MOOS, the modules are distinct applications each running with its own process ID on board the vehicle's computer. With IvP, the modules are distinct behaviors of the IvP Helm, which is a single MOOS process. The IvP Helm is an implementation of a behavior-based architecture for autonomous decision making.

A primary goal of a modular architecture is code re-use. At the risk of stating the obvious, people can't re-use code if they don't know about its existence, know what it does, or know where to find it. The purpose of this document is to address this issue. Many of the modules and topics discussed only briefly in this document are discussed in detail in other documents and we try to provide pointers to those resources as well.

## 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 utilities (seven of which are the topic of this document) and the IvP Helm and related behaviors. These modules are support on Linux and Mac OS X.

## 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 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.
- *Autonomy Behaviors of the IvP Helm Users Guide* - This document is a catalog and users manual for existing IvP Helm behaviors available on the moos-ivp.org website. It provides a detailed description of capabilities and interface specification for each behavior.

**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 Build Toolbox and General Guide for Writing New Behaviors for the IvP Helm* - This document is a users manual for those wishing to write their own IvP Helm behaviors. It describes the IvPBehavior superclass in detail. It also describes the IvPBuild Toolbox containing a number of tools for building IvP Functions which is the primary output of behaviors.

- *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.

- *MOOS-IvP Extensions - Extending a MOOS-IvP Autonomy System with New MOOS Applications and IvP Behaviors* - This document

## 2 Design Considerations of MOOS-IvP

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

### 2.1 Public Infrastructure - Layered Capabilities

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

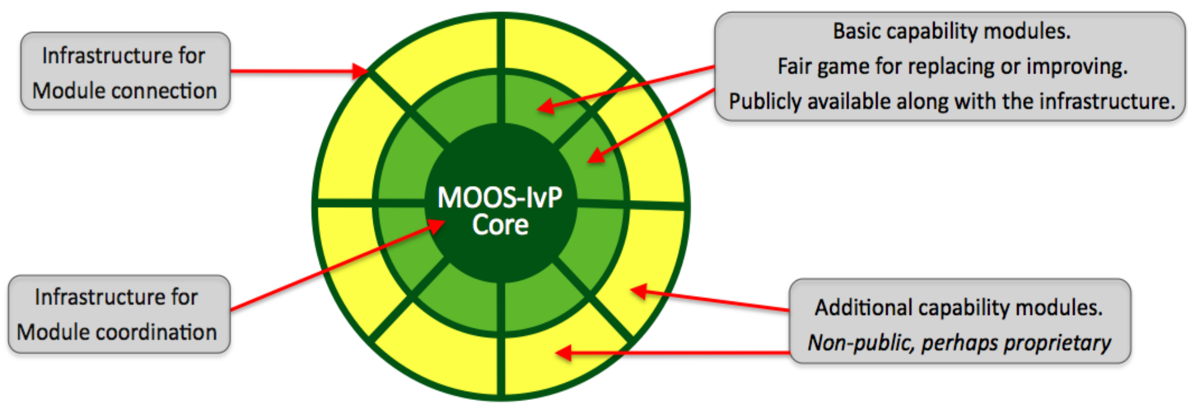


Figure 1: **Public Infrastructure - Layered Capabilities:** The center of the wheel represents MOOS-IvP Core. For MOOS this means the MOOSDB and the message passing and scheduling algorithms. For IvP this means the IvP helm behavior management and the multi-objective optimization solver. The wedges on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

The darker wedges in Figure 1 represent application modules (not infrastructure) that provide basic functionality and are publicly available. However, they do not hold any special immutable status. They can be replaced with a better version, or, since the source code is available, the code of the existing module can be changed or augmented to provide a better or different version (hopefully with a different name - see the section on branching below). Later sections provide an overview of about 40 or so particular modules that are currently available. By modules we mean MOOS applications and IvP behaviors and the above comments hold in either case. The 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 issues 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 needs 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.

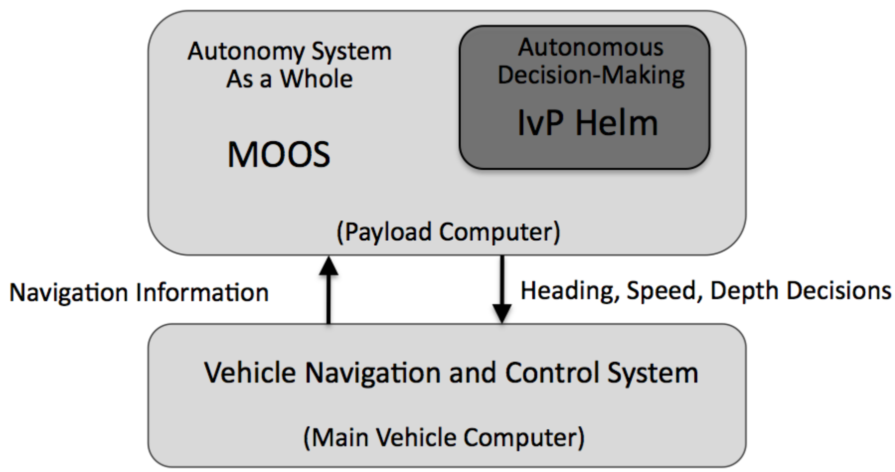


Figure 2: **The backseat driver paradigm**: The key idea is the separation of vehicle autonomy from vehicle control. The autonomy system provides heading, speed and depth commands to the vehicle control system. The vehicle control system executes the control and passes navigation information, e.g., position, 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.

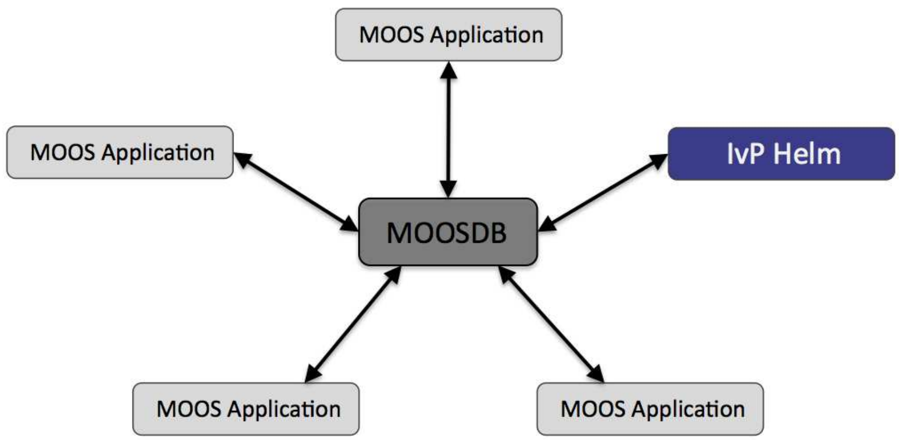


Figure 3: **A MOOS community**: is a collection of MOOS applications typically running on a single machine each with a separate process ID. Each process communicates through a single MOOS database process (the MOOSDB) in a publish-subscribe manner. Each process may be executing its inner-loop at a frequency 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).

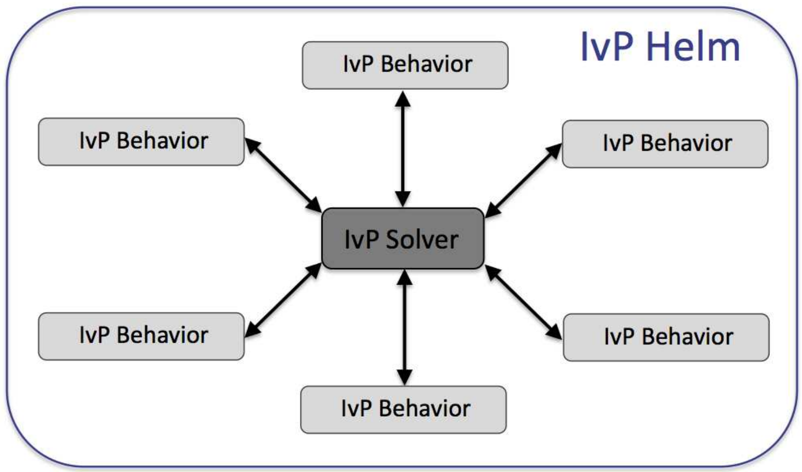


Figure 4: **The IvP Helm**: The helm is a single MOOS application running as the process pHelmIvP. It is a behavior-based architecture where the primary output of a behavior on each iteration is an IvP objective function. The IvP solver performs multi-objective optimization on the set of functions to 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 [6]. 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 [6]. Another approach is referred to as the potential fields, or vector summation approach (See [2], [8]) 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., [2], [3], [10], [11], land vehicles, e.g., [12], and marine vehicles, e.g., [5], [7], [9], [13], [14]. However, action-selection via the identification of a single highest priority behavior and via vector summation have well known shortcomings later described in [10], [11] and [12] 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 Module Management

The overall scope of this document is to provide, in a catalog style, an overview of software modules written under the umbrella term MOOS-IvP. By the term *module* we mean a group of C++ code that compiles into either an actual runnable application, or compiles into a library for linking into one or more applications. In this section a few issues regarding modules are addressed: (a) the difference between a library and application module, (b) module dependencies (c) systems integration and software acceptance, (d) rough assessments of module complexity, and (e) issues regarding branching the development of a module.

## 3.1 Library Modules Versus Application Modules

There are two types of repository modules discussed in this document, *libraries* and *applications*. When the source code of an application module is compiled it produces an application - an executable program that can be run by the user to perform a certain function. In this document most applications are MOOS applications, but some are not. When the source code of a library module is compiled it produces an archive - a piece of code capable of being used as a building block in one or more applications. The situation is depicted in Figure 5.

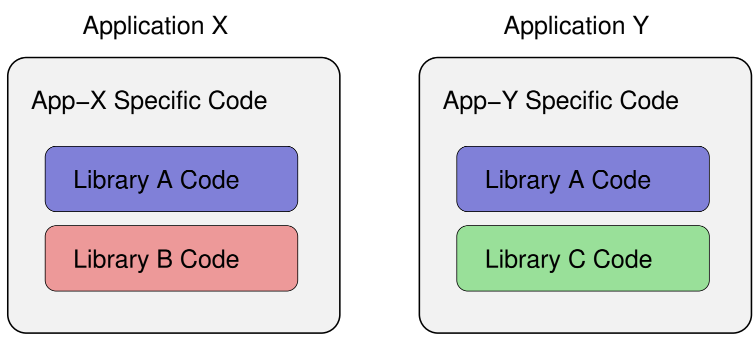


Figure 5: **Applications built with libraries**: An application module may contain source code specific to the application and source code from library archives. A library, like Library A in this example, may contribute to one or more applications realizing a common method of achieving "code re-use".

Of course the developer of the library may be different than the developer of an application. Often they are the same however, and code is partitioned this way to improve code re-use and reduce the complexity of maintenance. In addition to libraries released under MOOS-IvP, the applications in this document utilize widely available libraries outside the scope of this document such as OpenGL, FLTK, and libtiff for many GUI applications, as well as many standard libraries of C++ and the STL (Standard Template Library).

## 3.2 Module Dependencies

Referring to the example of Figure 5, when an Application X uses a Library A we say that there exists a *dependency* between the modules. In this case this is a one-way dependency since the application depends on the library but not vice-versa. From a systems maintenance perspective this means that the developers of Application X need not coordinate with the developers of Library A when they make changes to their own code. On the other hand, the developers of Library A may need to carefully consider how code changes will affect applications that depend on it. There is a greater need to document the interface for Library A and to consider backward compatibility if the interface changes. This dependency may seem cumbersome, but the efficiency achieved through code re-use in this manner is absolutely enormous. Furthermore, if the developers of Application X decide not to be dependent on what is effectively third party code from their perspective, they have the option under the Open Source license to adopt Library A as their own and disregard any future changes the developers of Library A may release. In the case where Application X is *released*, this approach of course must follow the terms of the Open Source license. Alternatively, a developer of an application may simply seek a different vendor who can provide a new library that matches the interface of Library A. The overview in this document covers both application and library module from the perspective of facilitating the configuration of and maintenance of a system comprised of modules from different sources. For this reason, the tables in later sections list applications, authors and library dependencies.

## 3.3 Systems Integration

An autonomy system comprised of modules with different capabilities from different vendors requires attention to the issue of software acceptance and software integration. The idea is sketched out in Figure 6.

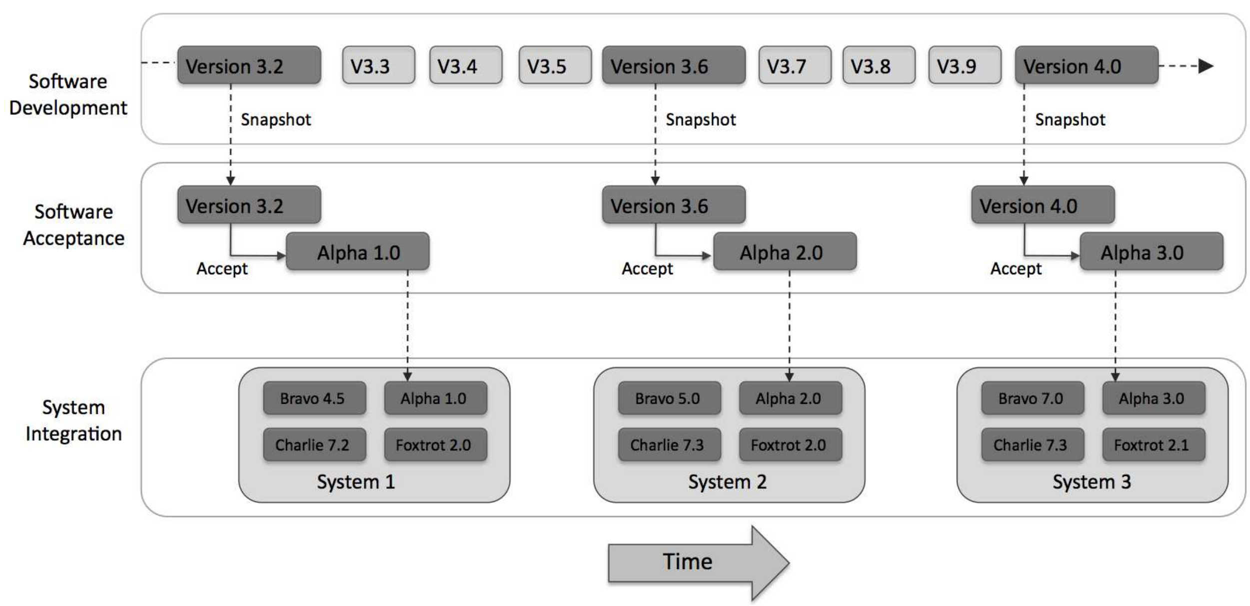


Figure 6: **Module Development, Acceptance, Integration**: The development cycle of a module includes software development and maintenance. Acceptance of a module by a third party may include the application of software test harnesses, human code review, or whatever the systems integrator may require. The accepted module may or may not adopt a unique name for integration purposes. Systems integration brings the accepted module into a larger group of modules perhaps from other sources or vendors dedicated to the readiness of a particular system, perhaps for a particular experiment or end-product release.

The roles of the software development, acceptance and integration are largely decoupled from one another beyond the idea that one serves as input to the other. Communication could be tightly coupled with feedback propagated back to the software developer, or completely decoupled, in the case of publicly released Open Source software anonymously accessed for a particular system and use.

### 3.4 Branching the Development of a Module

Software branching (see [1]) is "the duplication of an object under revision control (such as a source code file, or a directory tree) so that modifications can happen in parallel". Collaborating developers often branch to work separately on features before later merging their changes back into one work. But here we refer to the kind of branching that results in two distinct modules never to be merged again. There are a few reasons developers may choose to do this. Perhaps the original developer(s) no longer support further development at all, or the kinds of development of greatest interest to some set of users. Another reason may be that a set of users may wish to develop new features of the software that they are not interested in making available in a public release. Whatever the reason, the option of branching is a very powerful tool available to users that want to ensure that software continues to meet their evolving needs.

Each of the modules described in this document under an Open Source license are fair game for branching. This includes the modules later described as MOOS Core and IvP Core. The objective of course is to not branch on these core modules but to instead branch when necessary on the modules that plug into these cores - MOOS applications and IvP behaviors. Like most things, branching with moderation and deliberation is probably the best course. Here are a few things to consider in branching:

- If a branch is made, follow the terms of any applicable license.
- If a branch is made, call it something different. Some licenses mandate this, but it is a sensible practice appreciated by users who like to know what they're using, and to avoid giving the impression that the authors of a branch are taking undue credit.
- Only branch when necessary. A good rule of thumb is to engage the developers of the original software and see if the features you're interested in can be worked into a later release, or to offer help in adding the features by joining the development team. Although it's nice to have alternatives, having lots of alternatives with only small differences in capabilities is probably counter-productive.

.

### 3.5 Assessing Module Size and Complexity

In the tables in later sections, the size of a module is listed in terms of the number of lines of code. For library modules, this number is a direct line count applied to source code files. For application modules, two numbers are given. The first is the number of unique lines of code written specifically for the application and not shared by other applications. The second number includes additionally the line count of source code files from dependent libraries modules. The second number is always larger and is listed in bold face in the tables. The idea is depicted in Figure 7.

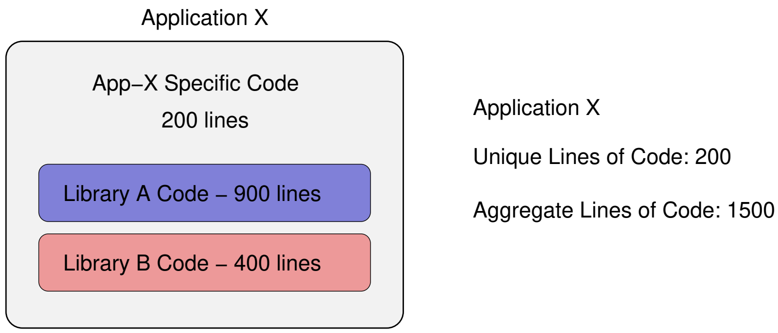


Figure 7: **Aggregate vs. unique lines of code**: The lines of code associated with an application in this document are listed in two groups. The *unique* lines of code are those written only for the application, and the *aggregate* lines are code include the unique line count plus the line counts from the libraries that contribute to the application.

In gauging the rough size and complexity of an application, which number should be used? There are pitfalls in using line counts of *any* type in gauging complexity or work effort. Some smaller pieces of code can certainly be more complex than some larger sets of code, and the same is true with gauging effort. Indeed often the product of a long concentrated coding effort can be *smaller* and cleaner code. Nevertheless, flawed as it may be, line counts do give some rough insight into the complexity and effort behind a module. As for the question of which line count is the more appropriate measure, the answer is subjective and probably some number between the two is best. The aggregate line count reflects the body of code involved overall in the compilation of the given application, regardless of how many other applications also make use of certain contributing libraries. However, the application may only be using a portion of the capabilities of a contributing library - in the extreme case just to bring in a single function available in a library with dozens of functions. On the other hand, using the unique line count as a reference, every line is likely to be directly involved in realizing the capability of the application. The unique line-count measure can be equally misleading in underestimating complexity as the aggregate line-count measure is in overestimating complexity. It is really case dependent, but some number in between is probably the best gauge.

# 4 The MOOS-IvP Software Modules

## 4.1 Overview

The MOOS-IvP software is comprised of modules served from both Oxford University and MIT/NUWC repositories. The Oxford project distributes the core MOOS implementation and several optional but widely used MOOS applications. The MIT MOOS-IvP project re-distributes certain MOOS modules from Oxford, augmented by several autonomy related MOOS applications and the core IvP Helm implementation developed at MIT and NUWC. This relationship is shown in Figure 8.

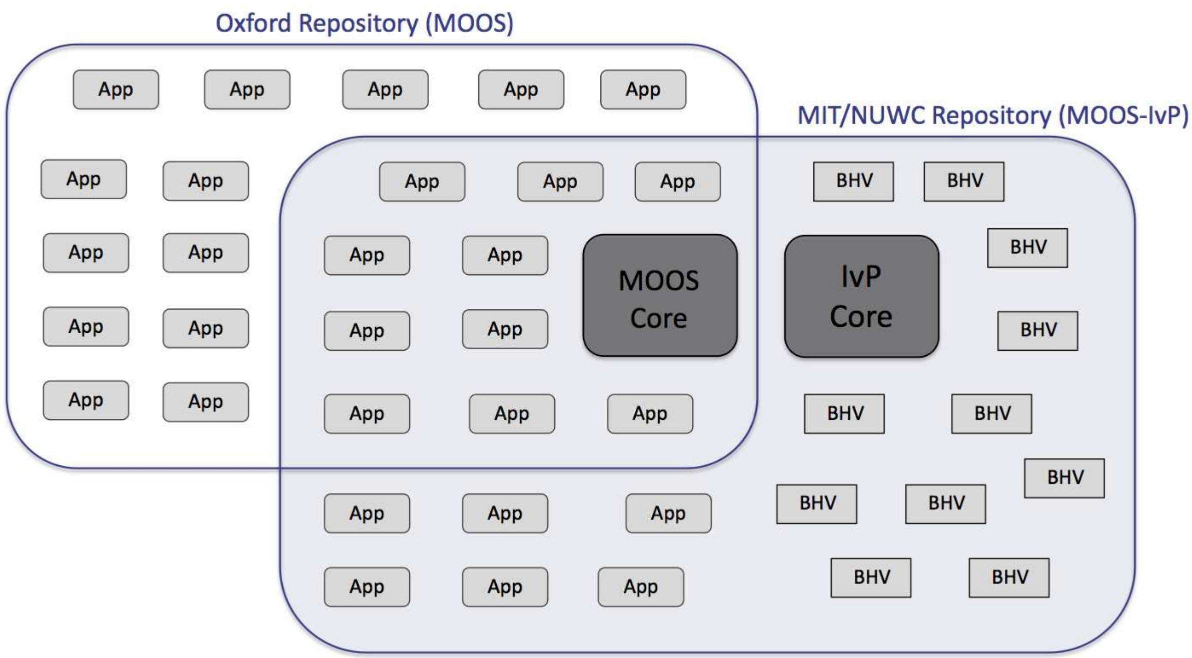


Figure 8: **The MOOS-IvP Repository**: The MOOS-IvP public codebase consists of software from both the Oxford University and MIT/NUWC repositories. The Oxford repository consists of the *MOOS Core* modules, and several further MOOS application modules. The MIT/NUWC repository consists of further MOOS modules developed by MIT/NUWC, the *IvPHelm Core* modules, and standard vehicle autonomy behavior modules. The modules labeled `App` are MOOS applications, and those labeled `BHV` are IvP Helm behavior modules.

The Oxford MOOS applications excluded from the MOOS-IvP server may have been excluded either because they are not publicly available from the Oxford Mobile Robotics Group, or they are not related to marine vehicles. The URL for the Oxford server is:

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

The URL for the MIT server is:

```
http://oceanai.mit.edu/moosivp
```

In the next few sections, specific modules making up portions of the diagram in Figure 3 are listed and briefly described.

### 4.2 The Oxford Software Repository

The Oxford repository contains two groups of software modules, (1) MOOS Core, and (2) other essential and common modules. The MOOS Core modules are listed in Table 1. The MOOS Essential modules are listed in Table 2.

#### 4.2.1 MOOS Core - Basic Middleware

The MOOS Core modules consist of MOOSDB, MOOSLIB, MOOSGenLib. These are considered "core" because they are the least common denominator between user communities wishing to use each other's modules interchangeably.

| # | Module Name | Module Description | Author | Size |
|---|---|---|---|---|
| 1 | **MOOSLIB** | A library of core MOOS data structures and application interfaces | Newman | 7,783 |
| 2 | **MOOSGenLib** | A library of tools for MOOS users | Newman | 6,596 |
| 3 | **MOOSDB** | The MOOS database application for handling the publish-subscribe architecture<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 2,351<br>**16,730** |
| | | Total unique lines of code | | 16,730 |
| | | Total aggregate lines of code | | **16,730** |

Table 1: Oxford MOOS Core Libraries

**MOOSDB:** The MOOSDB application is the single process at the heart of a running MOOS community (Figure 3). It is a single process that continuously communicates with connected MOOS applications accepting messages from applications as published data and distributing messages back to applications as subscribed data.

**MOOSLIB, MOOSGenLib:** The two libraries, MOOSLIB and MOOSGenLib, are linked against for virtually all known MOOS applications authored either at Oxford, MIT, NUWC or elsewhere. All MOOS applications define and implement a C++ class that is a subclass of a general MOOS application class. This superclass defines basic functionality for publishing and subscribing for information from a running MOOSDB process. These libraries define this superclass and other core functionality for inter-process communication over Sockets and TCP/IP as well as MOOS process scheduling, and mission file parsing.

### 4.2.2 Commonly Used MOOS Utility Applications

The tools described in this section are essential and common tools used by many MOOS users. They can however be regarded as options for a user and indeed other developers have created alternative MOOS applications with similar function to suit their purposes.

| # | Module Name | Module Description | Author | Size |
|---|---|---|---|---|
| 1 | **MOOSUtility** | A library of Marine-Vehicle data structures and utilities | Newman | 3,134 |
| 2 | **pAntler** | A script utility for launching a MOOS community of applications in concert<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 1,796<br>**16,175** |
| 3 | **pLogger** | A MOOS application for logging the MOOSDB transactions for later mission analysis<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 1,425<br>**15,804** |
| 4 | **pMOOSBridge** | A MOOS application for communication between MOOS communities<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 703<br>**15,082** |
| 5 | **pScheduler** | A MOOS application for scheduling variable-value postings to a MOOSDB<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 869<br>**15,248** |
| 6 | **uMS** | A GUI-based MOOS Scope for monitoring a running MOOSDB<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 902<br>**17,808** |
| 7 | **uPlayback** | A GUI-based tool for replaying logged data into a running MOOSDB<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 1,869<br>**16,248** |
| 8 | **iMatlab** | A MOOS application for linking live to running Matlab applications<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 1,403<br>**15,782** |
| 9 | **iRemote** | A platform remote-control interface to running MOOSDB<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 1,929<br>**1,929** |
| 10 | **uMVS** | A MOOS multi-vehicle simulator for marine vehicles<br>*Libraries: MOOSLIB, MOOSGenLib* | Newman | 4,145<br>**18,524** |
| | | Total unique lines of code | | 19,102 |
| | | Total aggregate lines of code | | **145,379** |

Table 2: Additional Oxford MOOS Modules

**MOOSUtilityLib:** A library of utilities used by many applications. It includes the very handy MOOSGeodesy algorithms for converting between Lat/Lon earth coordinates and local X/Y coordinates.

**pAntler:** The process pAntler is used to launch a MOOS community. One of the ideas underlying MOOS is the one mission file, one mission paradigm. A single mission file contains all the information required to configure all the processes needed to undertake the task (mission) in hand. Note a collection of MOOS processes is commonly referred to as a community. Antler provides a simple and compact way to start a MOOS mission, for example by typing `pAntler Mission.moos` on the command line.

**pLogger:** The `pLogger` process is intended to record the activities of a MOOS session. It can be configured to record a fraction of, or all publications of any number of MOOS variables. It is an essential MOOS tool and is worth its weight in gold in terms of post-mission analysis, data gathering and post-mission replay.

**pMOOSBridge:** The `pMOOSBridge` process is a powerful tool in building MOOS derived systems. It allows messages to pass between communities and is able to rename the messages as they are shuffled between communities. One instance of `pMOOSBridge` can "talk" to a limitless number of communities. The configuration block specifies what should be mapped or "shared" between communities and how it should be done. The `SHARE` command specifies precisely what variables should be shared between which communities.

**pScheduler:** The `pScheduler` process is a simple tool for generating and responding to messages sent to the MOOSDB by processes in a MOOS community. It supports three competencies - sequences, timers, and responses. A *Sequence* will publish given variable-value pair a given number of times with a given time interval. A *Timer* allows a variable to be written to the data base repetitively. A timer can be started and stopped by publication (by some other application) of user specified variables. The scheduler can also be told to derive the value of the periodic variable from another MOOS variable. A *Response* is used by the scheduler to respond to the publication of one variable with the publication of another.

**uMS:** The `uMS` application allows a user to place a stethoscope on the MOOS network and watch what variables are being written, which processes are writing them and how often this is happening. After starting up the scope and specifying the host name and port number of the MOOSDB the user is presented with a list of all MOOS variable in the server and their current state. Several times a second `uMS` calls into the DB and uses a special/unusual (and intentionally undocumented) message that requests that the server inform the client about all variables currently stored along with their update statistics. `uMS` is a central tool in the MOOS suite. It is cross platform and should be used to spy on and present visual feedback on any MOOS community.

**uPlayback:** The `uPlayback` application is a FLTK-based, cross platform GUI application that can load in alog files and replay them into a MOOS community as though the originators of the data were really running and issuing notifications. A typical use of this application is to fake the presence of sensor processes when reprocessing sensor data and tuning navigation filters. Alternatively it can be used in pure replay mode perhaps to render a movie of the recorded mission. The GUI allows the selection of which processes are faked. Only data recorded from those applications will be replayed from the log files.

**iMatlab:** The `iMatlab` application 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.

**iRemote:** The `iRemote` process was designed to be a control terminal for a deployed vehicle. It is really nothing more than a long switch statement based on characters input from the keyboard. One of its many functions is to allow remote control of the actuators of the vehicle. This is an invaluable asset for land and sub-sea vehicles alike. The application is multi-threaded. The primary thread blocks on a read of keyboard input. When a character is pressed some action is taken - for example publishing a new value for `DESIRED_THRUST`. The fact that `iRemote` can take control of a real vehicle presents a safety problem. What if the human controller walks away or even worse the vehicle moves out of communication range (eg a submarine dives) and the console is not available? To prevent the last issued actuator command being carried out ad-infinitum a secondary thread in iRemote prompts the user to hit an acknowledge key at least every 15 seconds. If the human driver does not respond the all actuators are set to the zero position.

**uMVS:** The `uMVS` process is a multi-vehicle AUV simulator. It is 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. When they intersect with acoustic devices (either on beacons or vehicles) the true time of intersection is calculated by a refinement process. This design allows the real round trip to be calculated when the vehicle undertakes a trajectory that was not known at the time the initial ping was launched.

### 4.3 The MIT/NUWC Software Repository

The MIT/NUWC software repository contains the IvP Helm and standard set of behaviors and utilities (Table 3), a set of MOOS utility applications (Table 4), and set of non-MOOS off-line utility applications (Table 5).

#### 4.3.1 IvP Core - Basic Autonomy Decision-Making

| # | Module Name | Module Description | Author | Size |
|---|---|---|---|---|
| 1 | **lib_ivpcore** | A library of core IvP data structures and solution algorithms | Benjamin | 4,505 |
| 2 | **lib_ivpbuild** | A library of tools for building IvP functions and evaluating their accuracy | Benjamin | 8,534 |
| 3 | **lib_bhvutil** | A library of marine autonomy behavior utilities | Benjamin | 2,146 |
| 4 | **lib_mbutil** | A library of general purpose utility tools | Benjamin | 2,527 |
| 5 | **lib_logic** | A library tools for implementing general purpose logic engines | Benjamin | 684 |
| 6 | **lib_behaviors** | A library defining general IvP behavior properties | Benjamin | 1,985 |
| 7 | **lib_geometry** | A library of geometry data structures and algorithms used by behaviors and GUI apps | Benjamin | 8,348 |
| 8 | **lib_behaviors-marine** | A library of marine autonomy behaviors | Benjamin | 7,558 |
| 9 | **lib_helmivp** | A library of the non-MOOS components of the IvP Helm | Benjamin | 1,429 |
| 10 | **pHelmIvP** | An autonomous helm for marine robotics<br>*Libraries: helmivp, behaviors-marine, bhvutil, behaviors, mbutil, ivpbuild, ivpcore, geometry, logic, MOOSLIB, MOOSGenLib* | Benjamin | 842<br>**51,508** |
| | | Total unique lines of code | | 38,558 |
| | | Total aggregate lines of code | | **51,508** |

Table 3: IvP core modules for implementing the IvP Helm and utilities used by the helm, and several other separate applications.

**lib_ivpcore:** The data structures defining the interval programming model, and the solutions algorithms are implemented in this module. This module is completely stand-alone with no use of code from other modules. If there is a true core to the IvP Helm software it would be this module.

**lib_ivpbuild:** Algorithms for the efficient creation of piecewise linearly defined (IvP) objective functions. This module is sometimes referred to as the IvPBuild Toolbox. It is an essential component to virtually any implemented IvP behavior for the purpose of creating the behavior objective function output.

**lib_mbutil:** This library is a mixed bag of utility functions of general use to many or nearly all applications and behaviors described in this section.

**lib_behaviors:** This library contains the IvP Behavior superclass defining the general syntactic structure inherited by all behaviors. It also defines core data structures used by the helm for posting information from behaviors to the MOOSDB.

**lib_bvhutil:** This library contains the software and algorithms shared between multiple behaviors and application modules that are used for debugging behaviors. For example, the WaypointEngine is defined here used by the Waypoint behavior, the Loiter behavior, and 3rd-party behaviors by other authors.

**lib_geometry:** The geometry library defines several geometric data structures and related algorithms for use in both autonomy behaviors as well as GUI-based tools that render geometric objects. Structures for segment lists, polygons, grids, points, lines, linear extrapolation are defined as well as algorithms for intersection tests, inter-object containment and sorting are defined here. For many behaviors, a substantial component of a behavior's complexity is derived from the algorithms in this library.

**lib_behaviors-marine:** All publicly available IvP Helm behaviors (Table 6) are implemented in this module. The full capabilities of these behaviors also draw part of their functionality from lib_geometry, lib_mbutil, lib_logic, and lib_ivpbuild.

**lib_helmivp:** This library contains the high-level IvP Helm implementation in the HelmEngine class. It also contains the mechanism for converting string configuration requests (from a behavior file) into actual behavior instantiations during helm initialization.

**pHelmIvP:** This is the actual IvP Helm application, but is basically a MOOS wrapper around the HelmEngine defined in lib_helmivp. It is kept relatively thin so the IvP Helm could easily be moved to some other publish-subscribe middleware without much effort.

### 4.3.2 Commonly Used MOOS Utility Applications

**iMarineSim:** The `iMarineSim` application is a very simplified vehicle simulator that updates vehicle state based on present actuator values. It runs locally in the MOOS community associated with the simulated vehicle, so there is one iMarineSim process running per each vehicle. Each iteration, after noting the changes in the navigation and actuator values, it posts a new set of navigation state variables

**pEchoVar:** The `pEchoVar` application is a lightweight process that runs without any user interaction for "echoing" the posting of specified variable-value pairs with a follow-on posting having different variable name. For example the posting of `FOO = 5.5` could be echoed such that `BAR = 5.5` immediately follows the first posting.

**pMarinePID:** The `pMarinePID` application provides simple PID control relating `DESIRED_SPEED` to `DESIRED_THRUST`, `DESIRED_HEADING` to `DESIRED_RUDDER`, and `DESIRED_DEPTH` to `DESIRED_PITCH`. The first value in each pair is produced typically by the helm, and the second value is subscribed to by various actuator drivers. This module is typically not part of set of autonomy modules when running in a "backseat" mode, since PID control is a "front seat" function. But it is used on some test vehicles such as the autonomous kayaks.

**pMarineViewer:** The `pMarineViewer` application is written with FLTK and OpenGL for rendering vehicles and associated op-area information and history during operation or simulation. In typical use, `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 AIS reports to the local community. The user is able to 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 to allow the interface to accept field control commands.

**pShipsideViewer:** The `pShipsideViewer` application is similar to the pMarineViewer application but assumes the user is operating a set of vehicles from a ship whose position information is also tracked and rendered on the geo display. The data fields on the viewer also give information relative to the deploying ship for each unmanned marine vehicle.

**uFunctionVis:** The `uFunctionVis` application renders objective functions produced by the IvP Helm behaviors. The display is updated automatically as behaviors post new functions. It only displays functions defined over *heading*, *speed*, or both. The display can be paused to temporarily not accept function updates, and the user can also choose to render the collective objective function - the weighted linear combination of functions produced by all behaviors for a given iteration. The function is rendered in 3D and the user can pan, rotate and zoom to facilitate analysis.

**uHelmScope:** The `uHelmScope` application is a console based tool for monitoring output of the IvPHelm, 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

| # | Module Name | Module Description | Author | Size |
|---|---|---|---|---|
| 1 | **lib_marineview** | A library of tools shared between GUI applications | Benjamin | 5,545 |
| 2 | **lib_navplot** | A library of tools for representing vehicle navigation plots | Benjamin | 1,471 |
| 3 | **lib_ipfview** | A library of graphics functions for rendering 3D objective functions | Benjamin | 1,066 |
| 4 | **iMarineSim** | A very simple marine vehicle simulator<br>*Libraries: mbutil, geometry, MOOSLIB, MOOSGenLib, MOOSUtility* | Benjamin | 1,039<br>**29,427** |
| 5 | **pEchoVar** | A tool for re-publising MOOS variables under a different name<br>*Libraries: mbutil, MOOSLIB, MOOSGenLib* | Benjamin | 329<br>**17,235** |
| 6 | **pMarinePID** | A PID controller for marine and land robots<br>*Libraries: mbutil, geometry, MOOSLIB, MOOSGenLib* | Benjamin, Newman | 1,387<br>**26,641** |
| 7 | **pMarineViewer** | A modest real-time 2D multi-vehicle renderer<br>*Libraries: marineview, mbutil, geometry, MOOSLIB, MOOSGenLib, MOOSUtility* | Benjamin | 1,547<br>**35,480** |
| 8 | **pShipsideViewer** | A ship-centric version of pMarineViewer with command and control<br>*Libraries: marineview, mbutil, geometry, MOOSLIB, MOOSGenLib, MOOSUtility* | Benjamin | 2,553<br>**36,486** |
| 9 | **pTransponderAIS** | An AIS-like report generator and receiver<br>*Libraries: mbutil, MOOSLIB, MOOSGenLib, MOOSUtility* | Benjamin | 901<br>**20,941** |
| 10 | **uFunctionVis** | A GUI for live rendering of IvP functions published by an IvP Helm<br>*Libraries: ipfview, geometry, ivpbuild, mbutil, ivpcore, MOOSLIB, MOOSGenLib* | Benjamin | 1,225<br>**40,584** |
| 11 | **uHelmScope** | A terminal-based scope on a running IvP Helm process<br>*Libraries: mbutil, MOOSLIB, MOOSGenLib* | Benjamin | 1,993<br>**18,899** |
| 12 | **uPokeDB** | A terminal based tool for poking the MOOSDB<br>*Libraries: mbutil, MOOSLIB, MOOSGenLib, MOOSUtility* | Benjamin | 340<br>**17,246** |
| 13 | **uProcessWatch** | A watch-dog process for monitoring a set of other MOOS processes<br>*Libraries: mbutil, MOOSLIB, MOOSGenLib* | Benjamin | 272<br>**17,178** |
| 14 | **uTermCommand** | A terminal based tool for pre-defined pokes of the MOOSDB<br>*Libraries: mbutil, MOOSLIB, MOOSGenLib, MOOSUtility* | Benjamin | 590<br>**17,496** |
| 15 | **uXMS** | A terminal based MOOS scope<br>*Libraries: mbutil, MOOSLIB, MOOSGenLib* | Benjamin | 902<br>**17,808** |
| | | Total unique lines of code | | 21,844 |
| | | Total aggregate lines of code | | **295,421** |

Table 4: Common MOOS applications from MIT/NUWC.

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.

**uPokeDB:** The `uPokeDB` application is a lightweight process that runs without any user interaction for writing to (poking) a running `MOOSDB` with one or more variable-value pairs. It is run from a console window with no GUI. It accepts the variable-value pairs from the command line, connects to the `MOOSDB`, displays the variable values prior to poking, performs the poke, displays the variable values after poking, and then disconnects from the `MOOSDB` and terminates. It also accepts a .moos file as a command line argument to grab the IP and port information to find the `MOOSDB` for connecting. Other than that, it does not read a `uPokeDB` configuration block from the `.moos` file.

**uProcessWatch:** The `uProcessWatch` application is a process for monitoring the presence of other MOOS processes, identified through the uProcessWatch configuration, to be present and connected to the MOOSDB under normal healthy operation. It does output a health report to the terminal, but typically is running without any terminal or GUI being display. The health report is summarized in two MOOS variables - `PROC_WATCH_SUMMARY` and `PROC_WATCH_EVENT`. The former is either set to "All Present", or is composed of a comma-separated list of missing processes identified as being AWOL (absent without leave). The `PROC_WATCH_EVENT` variable lists the last event affecting the summary, such as the re-emergence of an AWOL process or the disconnection of a process and thus new member on the AWOL list.

**uTermCommand:** The `uTermCommand` application is a terminal based tool for poking the MOOS database with pre-defined variable-value pairs. This can be used for command and control for example by setting variables in the MOOSDB that affect the behavior conditions running in the helm. One other way to do this, perhaps known to users of the iRemote process distributed with MOOS, is to use the Custom Keys feature by binding variable-value pairs to the numeric keys [0-9]. The primary drawback is the limitation to ten mappings, but the uTermCommand process also allows more meaningful easy-to-remember cues than the numeric keys.

**uXMS:** The `uXMS` application is a terminal based tool for live scoping on a MOOSDB process. Since it is not dependent on any graphic libraries it is more likely to run out-of-the-box on machines that may not have proper libraries like FLTK installed. It is easily configured from the command line or a MOOS configuration block to scope on as little as one variable. It can be run in a mode where screen updates only occur at the user's request. For these reasons, it is a good choice when bandwidth is an issue. It is also possible to have multiple versions of uXMS connected to the same MOOSDB. The uXMS tool was meant to be an alternative to the more feature-rich, high-bandwidth, GUI-based uMS process written and distributed on the Oxford MOOS website.

### 4.3.3 Commonly Used Off-Line Applications

The following are MOOS utility applications developed by MIT/NUWC complementing the MOOS utility applications distributed from Oxford.

| # | Module Name | Module Description | Author | Size |
|---|---|---|---|---|
| 1 | **lib_logutils** | A library of utilities for analyzing MOOS log files | Benjamin | 684 |
| 2 | **alogclip** | A tool for pruning MOOS alog files based on a given time window<br>*Libraries: mbutil* | Benjamin | 396<br>**2,923** |
| 3 | **alogscan** | A tool for scanning MOOS alog files and providing variable statistics<br>*Libraries: mbutil, logutils* | Benjamin | 220<br>**3,431** |
| 4 | **alogrm** | A tool for removing variable entries from a MOOS alog file by specifying variables or variable patterns to be removed.<br>*Libraries: mbutil, logutils* | Benjamin | 258<br>**258** |
| 5 | **aloggrep** | A tool for retaining variable entries from a MOOS alog file by specifying variables or variable patterns to be retained.<br>*Libraries: mbutil, logutils* | Benjamin | 223<br>**223** |
| 6 | **geoquery** | A tool for correlating Lat/Lon coordinates with local coordinates<br>*Libraries: mbutil, MOOSUtility* | Benjamin, Newman | 444<br>**6,105** |
| 7 | **geoview** | A tool for plotting and rendering waypoints on a 2D geodisplay<br>*Libraries: mbutil, geometry, MOOSUtility* | Benjamin | 1,043<br>**15,052** |
| 8 | **logview** | A tool for rendering logged vehicle data on a 2D geodisplay<br>*Libraries: mbutil, navplot, marineview, geometry, MOOSUtility* | Benjamin | 2,007<br>**23,032** |
| 9 | **splug** | A tool for defining input files with macros and file inclusion calls<br>*Libraries: mbutil* | Benjamin | 329<br>**2,856** |
| | | Total unique lines of code | | 4,920 |
| | | Total aggregate lines of code | | **60,302** |

Table 5: Off-line (non-MOOS) applications from MIT/NUWC

**lib_logutils** The `lib_logutils` library contains data structures and algorithms shared between most of the alog utility applications listed below.

**alogclip:** The `alogclip` program will read in a given alog file (produced by the pLogger MOOS application) and clip the file based on a begin time and stop time. All other log entries in between remain unchanged. This is a useful tool when a logfile has a lengthy section when a vehicle was sitting idle either before or after a mission while generating plentiful but useless data.

**aloggrep:** The `aloggrep` program will take as input (a) a given alog file (produced by the pLogger MOOS application) and (b) a list of MOOS variables, and (c) the name of a new alog file. It will produce a new alog file by *retaining* all lines from the original alog file that match the list of variables given on the command line. General pattern matching is not supported, but it will accept variable patterns of the form *FOOBAR and FOOBAR* matching all variables that end and begin with FOOBAR respectively.

**alogrm:** The alogrm program will take as input (a) a given alog file (produced by the pLogger MOOS application) and (b) a list of MOOS variables, and (c) the name of a new alog file. It will produce a new alog file by *removing* all lines from the original alog file that match the list of variables given on the command line. General pattern matching is not supported, but it will accept variable patterns of the form *FOOBAR and FOOBAR* matching all variables that end and begin with FOOBAR respectively.

**alogscan:** The alogscan program will read in a given log file (produced by the pLogger MOOS application) and generate two sets of statistics on the data. First it will provide, for each variable logged, the number of lines, characters, first log entry, last log entry, and which MOOS applications that published the variable. Second, it will provide, for each MOOS application, the total number of logfile lines, characters and the percentage of lines and characters for all lines and characters in the file the application is responsible for. It is a useful tool for a quick check that processes were running and producing what they were supposed to, and for finding ways to reduce log file bloat.

**geoquery:** The geoquery program is a tool for correlating lat/lon earth coordinates with local x/y coordinates, given a datum (lat/lon position corresponding to 0,0). This program is just an interface wrapper around Newman's MOOSGeodesy tools found in the MOOSUtility library in the Oxford MOOS distribution.

**geoview:** The geoview program is GUI based tool written with FLTK and OpenGL for rendering an op area image (e.g. images from Google Maps), and creating and editing geometric objects such as waypoint sequences, op limit boxes, search grids and so on.

**logview:** The logview program is a GUI based tool written with FLTK and OpenGL for rendering an op area image (e.g. images from Google Maps), and rendering vehicle trajectories read from the alog file of one or more vehicles. The user can step forward or backward through time to see how the multi-vehicle scenario played out. The user can also plot one or two (numerical) MOOS variables along the bottom of the screen to, for example, see how the vehicle speed or depth changed as the scenario unfolded. The user may also play through time automatically with variable time steps and automatically record a screen capture at each step, for later compilation into a video.

**splug:** The splug utility is a program for reading an input file and producing an output file while expanding certain macros and include-files in the input file to produce the modified output file. Although it is a general-purpose tool, it was built to ease the management of MOOS files and behavior files when the use of multiple vehicles, simulated or real, require the maintenance of larger sets of files.

### 4.3.4 IvP Helm Behaviors

The following are publicly available behaviors of the IvP Helm. Their implementation is in part found in the the lib_behaviors-marine module as well as the lib_behaviors module where the IvP-Behavior superclass is defined.

| # | Behavior Name | Behavior Description | Author |
|---|---|---|---|
| 1 | **Attractor** | Behavior for attracting the vehicle to a minimum distance from a target. | Schmidt |
| 2 | **AvoidCollision** | Avoiding collision between ownship and a given contact | Benjamin |
| 3 | **ConstantDepth** | Maintain an underwater vehicle at constant depth | Benjamin |
| 4 | **ConstantHeading** | Maintain an vehicle at constant heading | Benjamin |
| 5 | **CutRange** | Reduce the range between ownship and a given contact | Benjamin |
| 6 | **GoToDepth** | Periodically go to and maintain a given depth | Benjamin |
| 7 | **Loiter** | Repeatedly traverse a set of points comprising a convex polygon | Benjamin |
| 8 | **MemoryTurnLimit** | Constrain the turn based on recent heading history | Benjamin |
| 9 | **OpRegion** | Keep the vehicle inside a geo operations area or else all stop | Benjamin |
| 10 | **PeriodicSpeed** | Periodically exert a preference for a given speed or speed range | Benjamin |
| 11 | **PeriodicSurface** | Periodically bring the vehicle to the surface for a period of time of for a parameterized event | Benjamin |
| 12 | **RubberBand** | Acts like a rubber band for maintaining vicinity to a stationpoint | Schmidt |
| 13 | **Shadow** | Shadow a given contact in terms of matching speed and heading | Benjamin |
| 14 | **StationKeep** | Maintain a given vehicle position within a certain tolerance | Benjamin |
| 15 | **TimeOut** | Post a helm all-stop after a certain time has been exceeded. | Benjamin |
| 16 | **Timer** | Post one or more MOOS postings after a certain time has been exceeded. This may be done repeatedly. | Benjamin |
| 17 | **Trail** | Keep a relative position with respect to a given contact. | Benjamin |
| 18 | **Waypoint** | Transit through a sequence of given waypoints. | Benjamin |

Table 6: IvP Helm Basic Behaviors

**The `Attractor` Behavior:** The `Attractor` behavior is intended for attracting the vehicle towards a contact, when the distance between the two are larger than a certain minimum distance. Inside this distance the vehicle will simply shadow the contact. Outside the shadow range, the behavior priority increases linearly up to a maximum range beyond which it is constant. This behavior is particularly useful for developing a tracking solution for a moving target using bearing measurements, where you want the vehicle to get close to the target, while at the same time maintaining a small uncertainty of the bearing estimate, e.g. as determined from an acoustic array.

**The `AvoidCollision` Behavior:** The `AvoidCollision` behavior will influence the vehicle with its objective function to avoid a collision with another given contact of known position and trajectory. The objective function is defined over the coupled, dependent variables heading and speed. It is based on a calculation of closest point of approach (CPA) for given candidate maneuvers with a utility function applied to the CPA values. It can be configured to be inactive outside a given range to the contact with a linearly growing priority as the range between vehicles narrows. The user can also configure the tolerance of candidate maneuvers with respect CPA thus allowing the vehicle to be very conservative with respect to interpreting the notion of a near collision.

**The `ConstantDepth` Behavior:** The `ConstantDepth` behavior produces a utility function over the vehicle depth parameter. The user can configure the utility function to have a preferred depth peak, or plateau of equally acceptable depths. Depth values in the utility function different than the specified preferred depth or range of depths can be configured to have a linearly decreasing utility. Thus if other behaviors also have a depth preference, coordination/compromise will take place through the multi-objective optimization process.

**The `ConstantHeading` Behavior:** The `ConstantHeading` behavior produces a utility function over the vehicle heading parameter. The user can configure the utility function to have a preferred heading peak, or plateau of equally acceptable headings. Heading values in the utility function different than the specified preferred heading or range of headings can be configured to have a linearly decreasing utility. Thus if other behaviors also have a heading preference, coordination/compromise will take place through the multi-objective optimization process.

**The `ConstantSpeed` Behavior:** The `ConstantSpeed` behavior produces a utility function over the vehicle speed parameter. The user can configure the utility function to have a preferred speed peak, or plateau of equally acceptable speeds. Speed values in the utility function different than the specified preferred speed or range of speeds can be configured to have a linearly decreasing utility. Thus if other behaviors also have a speed preference, coordination/compromise will take place through the multi-objective optimization process.

**The `CutRange` Behavior:** The `CutRange` behavior will drive the vehicle to reduce the range between itself and another specified vehicle - nearly the opposite of the `AvoidCollision` behavior. It produces a utility function over the heading and speed dependent decision variables. It can operate in two modes (or a linear combination between the two), where in one mode the utility function is based on the calculated closest point of approach (CPA) of a candidate ownship maneuver and given contact position and track. In the other mode, the utility function ranks heading and speed decisions that drive ownship toward the contact as if it were stationary at its known position.

**The `GoToDepth` Behavior:** The `GoToDepth` 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 `Loiter` Behavior:** The `Loiter` behavior is used for transiting to and repeatedly traversing a set of waypoints. A similar effect can be achieved with the Waypoint behavior but this behavior requires set of waypoints form a convex polygon to exploit certain useful algorithms. This behavior utilizes the non-monotonic arrival criteria use in the Waypoint behavior to avoid loop-backs upon waypoint near-misses. It also robustly handles dynamic exit and re-entry modes when or if the vehicle diverges from the loiter region due to external events. As with all other behaviors, it can be dynamically reconfigured to allow a mission control module to repeatedly reassign the vehicle to different loiter regions by using a single persistent instance of the behavior. It can also be configured to loiter at the vehicle's present position when commanded to go into loiter mode.

**The `MemoryTurnLimit` Behavior:** The `MemoryTurnLimit` behavior influences the heading of the vehicle to discourage sharp turns that would allow it to come back around on its own track too quickly. It locally stores a history of ownship headings and is typically configured to discourage new heading values that deviate greatly from the heading history. This behavior was motivated by UUVs towing a hydrophone array and the desire not to twist the array with rapid vehicle turns.

**The `OpRegion` Behavior:** The `OpRegion` behavior does not produce an objective function but instead monitors certain safety conditions and sends a message to the helm to shut down (with a zero-speed command) if the conditions are violated. The primary condition is to ensure that ownship remains within a given operation area given by a convex polygon given by a set of vertices. Other conditions are a maximum overall mission time, a maximum vehicle depth, and a minimum vehicle altitude. This behavior does not influence the vehicle to avoid violating these limits but simply monitors the conditions, and raising a red flag if they are violated.

**The `PeriodicSpeed` Behavior:** The `PeriodicSpeed` 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 `PeriodicSurface`** The `PeriodicSurface` 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 `RubberBand` Behavior:** The `RubberBand` behavior is similar to the `Attractor` behavior, but with a different range-dependence of the priority. Thus, inside a minimum range the priority is zero, similar to the behavior of a relaxed rubber band. Once extended, the attraction force (i.e. the priority) increases linearly, as is the case for the `Attractor` behavior. Beyond the maximum range, the priority continues to grow linearly, but at a lower rate, similar to the elasto-plastic region for an extended rubber band. This behavior is useful for maintaining the vehicle within a certain operational area around it's station point, while still allowing it to be attracted to a target for generating tracking solutions. This behavior is therefore often applied together with the `Attractor` behavior.

**The `Shadow` Behavior:** The `Shadow` behavior will influence the vehicle with its objective function to mimic the observed heading and speed of a given known contact.

**The `StationKeep` Behavior:** The `StationKeep` behavior is designed to keep the vehicle at a given lat/lon or x,y 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 speed if the vehicle is outside the outer 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 station point can be set in a mission file, communicated by mission control, or set to be the current position when the behavior becomes active.

**The `Timer` Behavior:** This behavior can 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 event (with end flags). The `DURATION`, `CONDITION`, `RUNFLAG` and `ENDFLAG` parameters are all defined generally for behaviors. There are no additional parameters defined for this behavior.

**The `Trail` Behavior:** The `Trail` behavior will influence the vehicle with its objective function to trail or follow another specified vehicle at a given relative position. It can be used as a tool for "formation flying". It operates in two modes - one when the vehicle is outside a tolerance radius from the trail point, and one when the vehicle is inside. When inside the tolerance radius, the behavior acts like the `Shadow` behavior as described above. When outside the tolerance radius it acts like the `CutRange` behavior. The relative position to the contact is given by the *relative* bearing and range to a contact, but can also be set to work from the *absolute* bearing instead.

**The `Waypoint` Behavior:** The `Waypoint` behavior is used for transiting to a set of specified waypoints. The objective function produced by this behavior is defined over the 2D action space given by possible heading and speed choices. The behavior can be optionally configured to track-line follow the segment between any two waypoints or simply steer toward the next waypoint.

# References


[1] http://en.wikipedia.org/wiki/Branching_(software).

[2] 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.

[3] 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.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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