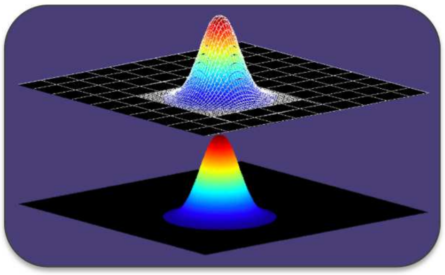
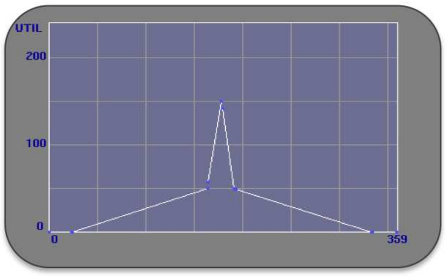
MIT-CSAIL-TR-2009-037 August 20, 2009

Extending a MOOS-IvP Autonomy System and Users Guide to the IvPBuild Toolbox

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

Extending a MOOS-IvP Autonomy System and Users Guide to the IvPBuild Toolbox





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, Code 2501
NUWC Division Newport, Newport RI

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

August 11th, 2009 - Release 4.0 beta (SVN Revision 2260)

Abstract

This document describes how to extend the suite of MOOS applications and IvP Helm behaviors distributed with the MOOS-IvP software bundle from www.moos-ivp.org. It covers (a) a straw-man repository with a place-holder MOOS application and IvP Behavior, with a working CMake build structure, (b) a brief overview of the MOOS application class with an example application, and (c) an overview of the IvP Behavior class with an example behavior, and (d) the IvPBuild Toolbox for generation of objective functions within behaviors.

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 document describes how to extend the set of modules beyond those distributed in the MOOS-IvP bundle from varwww.moos-ivp.org. It addresses the reader who is familiar with how to use MOOS applications and the IvP helm, but is interested in building their own MOOS application and/or IvP behavior. This document covers (a) a straw-man repository with a place-holder MOOS application and IvP Behavior, with a working CMake build structure, (b) a brief overview of the MOOS application class with an example application, and (c) an overview of the IvP Behavior class with an example behavior, and (d) the IvPBuild toolbox for generation of objective functions within behaviors.

This document is still in draft form and has known omissions. The reader is encouraged email the author(s) feedback at issues@moos-ivp.org, and to look for later versions on www.moos-ivp.org.

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/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.moos-ivp.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. MIT CSAIL Technical Report TR-2009-28.
- *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. MIT CSAIL Technical Report TR-2008-65.
- *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. MIT CSAIL Technical Report TR-2009-006.
- *Extending a MOOS-IvP Autonomy System and Users Guide to the IvPBuild Toolbox* (this document) - 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, the primary output of behaviors. It provides an example template directory with example IvP Helm behavior and an example MOOS application along 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 [4].

2 Extending MOOS-IvP By Example

2.1 Brief Overview

This section describes an example repository distributed with the MOOS-IvP software bundle at www.moos-ivp.org. This repository merely provides a template with an example MOOS application, IvP Behavior, and example mission. More importantly perhaps is that the CMake build files are provided. A cursory look at these files reveal the hooks to add a new behavior or application. This is meant to provide one easy way to begin extending the MOOS-IvP software capabilities with one's own modules.

2.2 Obtaining and Building the Example Extensions Folder

The example extensions folder is available at the following URL:

```
http://www.moos-ivp.org/software/extensions.html
```

Instructions are provided for downloading the software from an SVN server with anonymous read-only access. After checking out the tree from the SVN server as prescribed at this link, the top level directory should have the following structure:

```
moos-ivp-extend/
   bin/
   docs/
   missions/
   src/
```

The build instructions are maintained in the README files and are probably more up to date than this document. In short building the software amounts to two steps:

```
> cd moos-ivp-extend/src/
> cmake ./
> make
```

The build depends on the directory `moos-ivp-extend` being in the same directory as `moos-ivp`. If this needs to be different on your system, the file `CMakeLists.txt` in the `src/` directory can be edited. The relevant lines are at the top of the file:

```
GET_FILENAME_COMPONENT(MOOS_BASE_DIR_A    ../../moos-ivp/trunk/MOOS  ABSOLUTE)
GET_FILENAME_COMPONENT(IVP_BASE_DIR_A     ../../moos-ivp/trunk/ivp   ABSOLUTE)
GET_FILENAME_COMPONENT(MOOS_BASE_DIR_B    ../../moos-ivp/MOOS        ABSOLUTE)
GET_FILENAME_COMPONENT(IVP_BASE_DIR_B     ../../moos-ivp/ivp         ABSOLUTE)
```

After building the software there should be a new MOOS application called `pXRelayTest` in the `bin/` directory, and a new IvP Behavior in the directory `src/lib_behaviors-test/` directory. The new behavior is in the form of a shared object, having the name `libBHV_SimpleWaypoint.so` in Linux, and `libBHV_SimpleWaypoint.dylib` on the Mac OS X platform.

2.3 Using the New MOOS Application

To use the new MOOS application, the directory `moos-ivp-extend/bin/` needs to be added to the user's shell path. This is typically done in the `.cshrc` or `.bashrc` file for tcsh and bash users respectively. To confirm that things are ready to go, use the built-in shell command `which`:

```
> which pXRelayTest
```

which returns the directory where the executable resides if it is indeed in the shell's path. Otherwise it returns nothing. Don't forget that an edited path doesn't take effect until a new shell is launched or unless the user types `"source .cshrc"`, or `"source .bashrc"`.

The `pXRelayTest` application is the same as the `pXRelay` application distributed with the MOOS-IvP software bundle. It differs only in name for the sake of illustrating the process of building a new application outside the `moos-ivp` tree. This example MOOS application is described in detail in Section 3.8. In that section, an example mission file is described for running two `pXRelay` processes to illustrate their function. A similar mission file is provided in:

```
moos-ivp-extend/missions/xrelay/xrelay.moos
```

that launches two processes, `pXRelay` and `pXRelayTest` as a way of confirming that you are running a MOOS application from the extensions build alongside the build of the main moos-ivp repository. Information on how to work through this example is provided in Sections 3.8.2 and 3.8.3.

2.4 Using the New IvP Helm Behavior

To use the new IvP Helm behavior built in the extensions folder, the helm needs to know about it. The helm already contains a number of behaviors compiled in to the `pHelmIvP` executable, but the objective of adding behaviors in the manner outlined here, is to avoid any recompiling of the helm as new behaviors are added. Loosely speaking, there is a one-way dependency between repositories - new behaviors are layered onto the set of behaviors shipped with the helm with no modifications or re-build required of the basic moos-ivp software tree.

Newly built behaviors are compiled in to shared object files, `*.so` in Linux, and `*.dylib` in Mac OS X. The helm references a path variable called `IVP_BEHAVIOR_DIRS` which contains a colon-separated list of all directories containing dynamically loadable behaviors. This variable is a shell environment variable and is typically set in the `.cshrc` or `.bashrc` file for tcsh and bash users respectively. For example, the following line in the `.cshrc` file for tcsh users:

```
setenv IVP_BEHAVIOR_DIRS = '/home/bob/moos-ivp-extend/src/lib_behaviors-test'
```

A mission file to test this is provided in:

```
moos-ivp-extend/missions/alder/alder.moos
```

The mission is launched with:

```
> cd moos-ivp-extend/missions/alder/
> pAntler alder.moos
```

The output produced in the helm terminal window should look like that shown in Listing 1 below, and provides useful feedback on whether the dynamically loadable behavior was loaded properly.

Listing 1 - Example `pHelmIvP` *terminal output when loading a dynamic behavior.*

```
 0  ***************************************************
 1  *                                                 *
 2  *       This is MOOS Client                       *
 3  *       c. P Newman 2001                          *
 4  *                                                 *
 5  ***************************************************
 6
 7  ---------------MOOS CONNECT-----------------------
 8    contacting a MOOS server localhost:9000 -  try 00001
 9    Contact Made
10    Handshaking as "pHelmIvP"
11    Handshaking Complete
12    Invoking User OnConnect() callback...ok
13  --------------------------------------------------
14
15  The IvP Helm (pHelmIvP) is starting....
16  Loading behavior dynamic libraries....
17      Loading directory: /Users/mikerb/Research/moos-ivp-extend/src/lib_behaviors-test
18          About to load behavior library: BHV_SimpleWaypoint ... SUCCESS
19  Loading behavior dynamic libraries - FINISHED.
20  Number of behavior files: 1
21  Processing Behavior File: alder.bhv  START
22      Successfully found file: alder.bhv
23      InitializeBehavior: found dynamic behavior BHV_SimpleWaypoint
24      InitializeBehavior: found dynamic behavior BHV_SimpleWaypoint
25  Processing Behavior File: alder.bhv  END
26  mode description:
27  pHelmIvP is Running:
28           AppTick   @ 4.0 Hz
29           CommsTick @ 4 Hz
```

The output prior to line 15 is standard MOOS output for an application connecting to the MOOSDB server. The lines thereafter are specific to the `pHelmIvP` application. In lines 16-19, the helm indicates that the directories specified in the `IVP_BEHAVIOR_DIRS` environment variable were found and indicates all dynamic behaviors loaded from those directories, regardless of whether they are used in this mission. Line 20 indicates the number of behavior files (`.bhv` files) comprising this mission. For each behavior file, output similar to lines 21-26 are generated which reports on the attempts to load individual behavior, noting for each whether they are a static behavior of a dynamically loaded behavior.

When the example is fully launched, the `pMarineViewer` should appear with a simulated vehicle, and two buttons at the lower right corner. The vehicle can be launched by clicking the "DEPLOY" button. The dynamically loaded behavior is called `BHV_SimpleWaypoint` and is described in detail in Section 5.

2.5 Extending the Extensions

To add further MOOS application modules, the simplest way by this example is to create sibling directories to the `pXRelayTest`, and add the corresponding entry to the `CMakeLists.txt` file in the `src/` directory. Further IvP behaviors can be added within the `lib_behaviors-test` directory, or in a separate `lib_*` directory. In the former case, the `CMakeLists.txt` file in the behavior directory needs to be augmented for the new behavior. In the latter case, an extra entry in the `CMakeLists.txt` file in the `src/` directory is required, as well as the addition of another directory in the `IVP_BEHAVIOR_DIRS` variable as described above in Section 2.4.

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 [13].

3.1 Inter-process communication with Publish/Subscribe

MOOS has a star-like topology. 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.

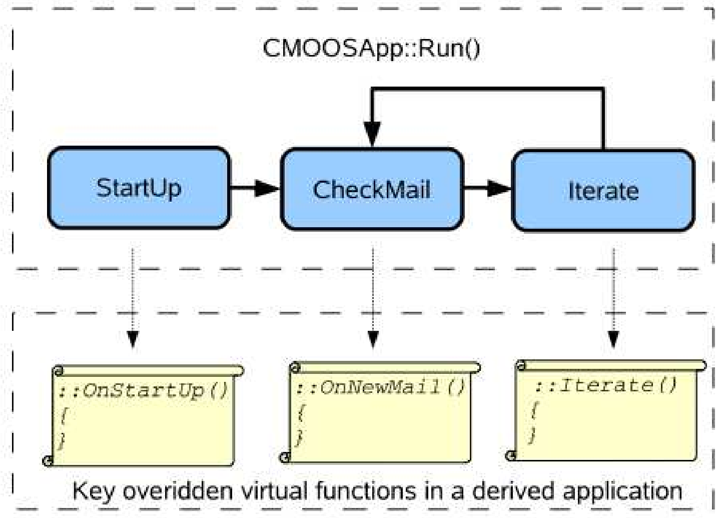


Figure 1: **Key virtual functions of the MOOS application base class**: The flow of execution once `Run()` has been called on a class derived from `CMOOSApp` . The scrolls indicate where users of the functionality of CMOOSApp will be writing new code that implements whatever it is that is wanted from the new applications.

The roles of the three virtual functions in Figure 1 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 [13].

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 1. 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.

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 5-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 7. 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 2 - 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 [3], 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 2) which confirm that the `Iterate()` loop in each process is indeed being executed. The output for the other variables in Listing 2 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 3 - 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 [3]. 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 4:

Listing 4 - 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 5-A through 5-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 2 above.

Listing 5-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 5-B below, the two `pXRelay` applications are configured. Note that the argument to `ProcessConfig` on lines 20 and 32 is the alias for `pXRelay` specified in the Antler configuration block on lines 12 and 13. Each `pXRelay` process is configured such that its incoming and outgoing MOOS variables complement one another on lines 25-26 and 37-38. Note the `AppTick` parameter (see Section 3.4.1) is set to 25 in both configuration blocks, and compare with the observed frequency of the `Iterate()` function reported in the variables `APPLES_ITER_HZ` and `PEARS_ITER_HZ` in Listing 2. MOOS has done a pretty faithful job in this example of honoring the requested frequency of the `Iterate()` loop in each application.

Listing 5-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 5-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 2 and 4. 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 [3] for further ways to configure the `uXMS` tool.

Listing 5-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 the `pXRelay` configuration blocks in the `xrelay.moos` file, re-start, and note the resulting change in the iteration and post frequencies in the `uXMS` output.
- Try changing the `CommsTick` parameter in *one* of the `pXRelay` configuration blocks in the `xrelay.moos` file to something much lower than the `AppTick` parameter, re-start, and note the resulting change in the iteration and post frequencies in the `uXMS` output.

3.9 MOOS Applications Available to the Public

Below are very brief descriptions of MOOS applications in the public domain. This is by no means a complete list. It does not include applications outside MIT, 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 [5].

3.9.1 MOOS Modules from Oxford

- `pAntler:` A tool for launching a collection of MOOS processes given a mission file. See [13], [12]. .
- `pMOOSBridge:` A tool that allows messages to pass between communities and allows for the renaming of messages as they are shuffled between communities. See [13], [12].
- `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 [5], [12].
- `pScheduler:` A simple tool for generating and responding to messages sent to the MOOSDB by processes in a MOOS community. See [5], [12].
- `uMS:` A GUI-Based MOOS scope for monitoring one or more MOOSDBs. See [5], [12].
- `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 [5], [12].

- `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 [5], [12].
- `iRemote`: A terminal-based tool for remote control of a robotic platform running MOOS. It can be configured to associate a pre-defined variable-value poke with any un-mapped key on the keyboard. See [5], [12].
- `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 [5], [12].

3.9.2 MOOS Modules from MIT and NUWC

- `pHelmIvP`: The IvP Helm, and primary focus of this document.
- `pNodeReporter`: The `pNodeReporter` application garners vehicle navigation information such as position, speed, heading, yaw and depth, along with high-level helm information such as its operation mode, and publishes a summary variable, `NODE_REPORT_LOCAL`, which is consumed by viewer applications such as `pMarineViewer`, and as input to other vehicles participating in cooperative tasks.
- `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 [3], [7].
- `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 [3], [7].
- `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 [3], [7].
- `uXMS`: A terminal based tool for live scoping on a MOOSDB process. See [3], [7]. .
- `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 [3], [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 [3], [7].
- `uTimerScript:` A MOOS application that will poke the MOOSDB with pre-defined variable-value pairs in a script that may repeat. Not unlike `pScheduler`, but it can do some additional things such as jump forward or pause in the script based on MOOS notifications. It may also schedule its events to occur at a random point in a fixed time interval.See [3], [7].

4 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.

4.1 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 2. 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.

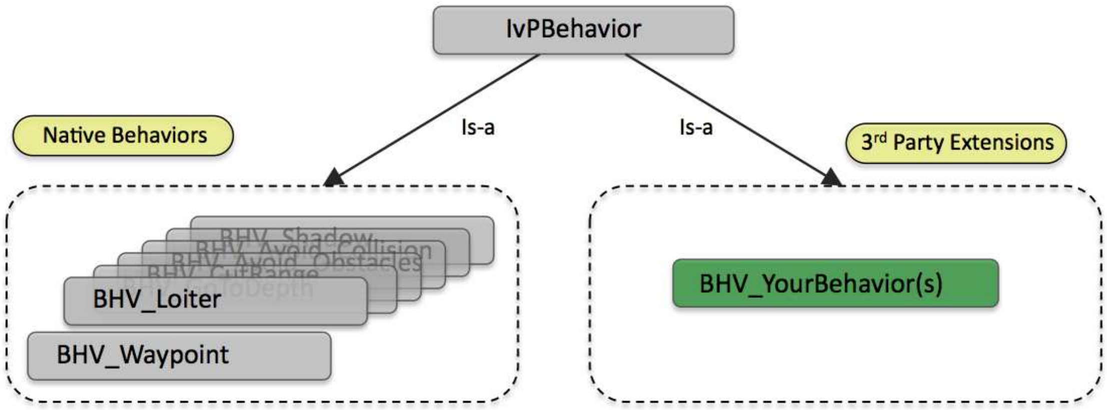


Figure 2: **Behavior inheritance**: Behaviors are derived from the `IvPBehavior` superclass. The native behaviors are the behaviors distributed with the helm. New behaviors also need to be subclass of the IvPBehavior class to work with the helm. Certain virtual functions invoked by the helm may be optionally 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.

4.2 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 [6] in the section "IvP Helm Autonomy" 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.

4.2.1 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 4.2.3 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 4.2.3 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 4.2.3 for more detail.

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

`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 4.2.3 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 the section "Behavior Run Conditions" in [6] for more detail.

`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 the section "Behavior Flags and Behavior Messages" in [6] for more detail.

`IDLEFLAG:` This parameter specifies a variable and a value to be posted when the behavior is in the *idle* state. See the section "Behavior Run States" in [6] for more detail 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 the section "Behavior Flags and Behavior Messages" in [6] for more detail.

`ACTIVEFlAG:` This parameter specifies a variable and a value to be posted when the behavior is in the *active* state. See the section "Behavior Run States" in [6] for more detail 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 the section "Behavior Flags and Behavior Messages" in [6] for more detail.

`INACTIVEFlAG:` This parameter specifies a variable and a value to be posted when the behavior is *not* in the *active* state. See the section "Behavior Run States" in [6] for more detail 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 the section "Behavior Flags and Behavior Messages" in [6] for more detail.

`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 the section "Behavior Flags and Behavior Messages" in [6] for more detail.

`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 4.2.2 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 4.2.5 on page 33 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 4.2.4 for more detail.

4.2.2 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 6. 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 6 - 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 6). 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.

4.2.3 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.

4.2.4 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 the section "Behavior Flags and Behavior Messages" in [6] for more detail. 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.

4.2.5 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.

4.3 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 4.2) handled in the superclass version of this function:

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

The `setParam()` function should return `true` if the parameter is recognized and the value is in an acceptable form. In the rare case that a new behavior has no additional parameters, leaving this function undefined in the subclass is appropriate. The example below in Listing 7 gives an example for a fictional behavior `BHV_YourBehavior` having a single parameter `period`.

Listing 7 - 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.

4.4 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 3. 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 3:

- 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.

4.4.1 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 3 regarding the sequence of these function calls.

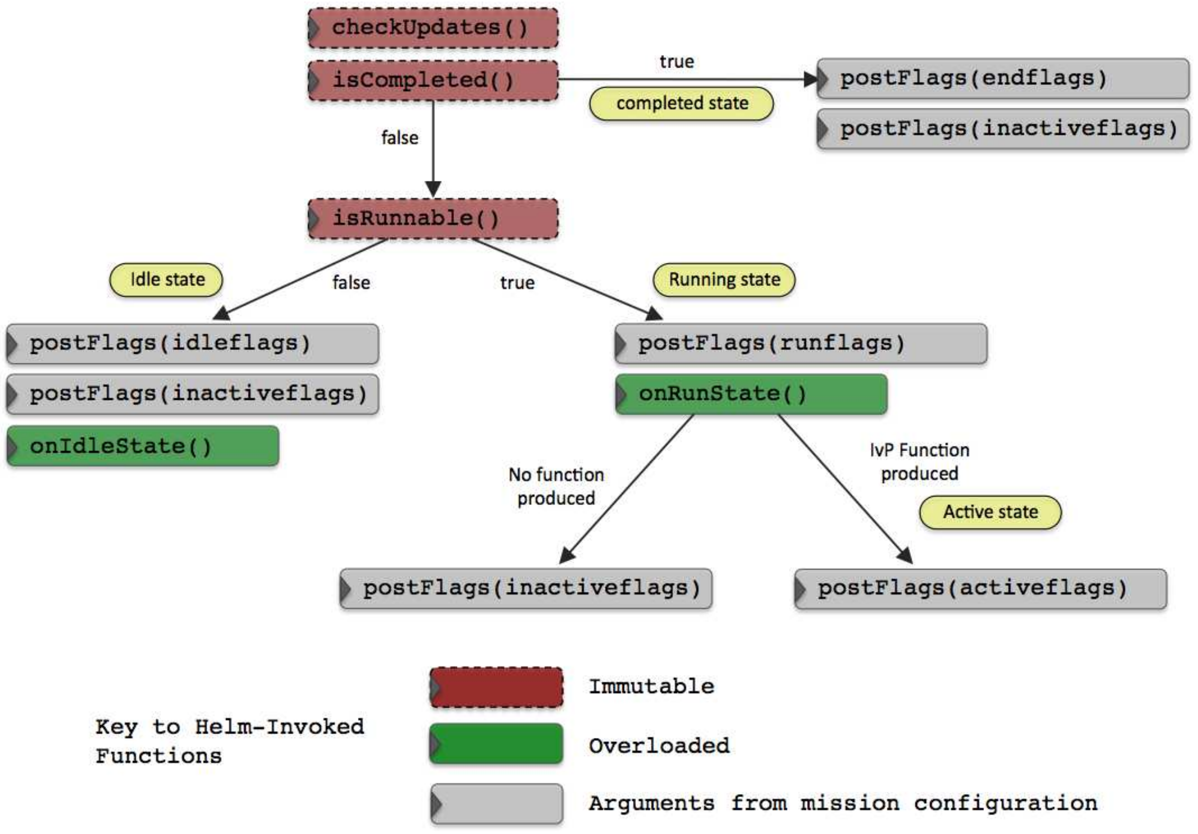


Figure 3: **Behavior function-calls by the helm**: The helm invokes a sequence of functions on each behavior on each iteration of the helm. The sequence of calls is dependent on what the behavior returns, and reflects the behaviors activity state. Certain functions are immutable and can not be overloaded 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 4.2.2 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 4.2.5). (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 the section "Behavior Run Conditions" in [6] 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 the section "Behavior Flags and Behavior Messages" in [6] for more detail on flag types.

4.4.2 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 3). The bulk of the work in implementing a new behavior is in this function implementation, and is the subject of Section 4.6.

`void onIdleState()`: This function is called by the helm when deemed to be in the *idle* state (Figure 3). 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.

4.5 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.

4.5.1 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 4.5.3.

`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 4.5.2.

`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 4.5.2.

`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 4.5.2.

`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 4.5.2.

`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 4.5.2.

`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 the section "Automated Filtering of Successive Duplicate Helm Publications" in [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 the section "Automated Filtering of Successive Duplicate Helm Publications" in [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 the section "Automated Filtering of Successive Duplicate Helm Publications" in [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 the section "Automated Filtering of Successive Duplicate Helm Publications" in [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.

4.5.2 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.

4.5.3 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.

4.5.4 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.

4.6 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 3 on page 35. 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 4.5.2 - 4.5.4. The functions for posting messages to the MOOSDB from within a behavior were discussed in Section 4.5.1. Further issues regarding the posting of messages were covered in the section "Automated Filtering of Duplicate Helm Publications" in [6]. The remaining issue to discuss is how objective functions are generated. This is covered in the IvPBuild Toolbox, in Sections 6, 7, 8, and 9.

5 An Implementation Example - the SimpleWaypoint Behavior

In this section an example IvP behavior is presented. It is a simplified waypoint behavior version of the waypoint behavior in the standard suite of behaviors distributed with the MOOS-IvP public software bundle. The class name for this behavior is `BHV_SimpleWaypoint`. This behavior is distributed in the "moos-ivp-extend" repository and should build out of the box. After going through the class itself, later in this section example missions, also distributed with "moos-ivp-extend", for running the behavior are discussed.

5.1 The SimpleWaypoint Behavior Class Definition

The SimpleWaypoint behavior is configured with four parameters: a single waypoint given in terms of local x and y coordinates, a transit speed in meters per second, and a radius in meters around the destination point within which the vehicle will be declared to have arrived at its waypoint. The behavior, at every iteration of the helm loop, notes the vehicle's own position in x and y local coordinates. The idea is shown in Figure 4.

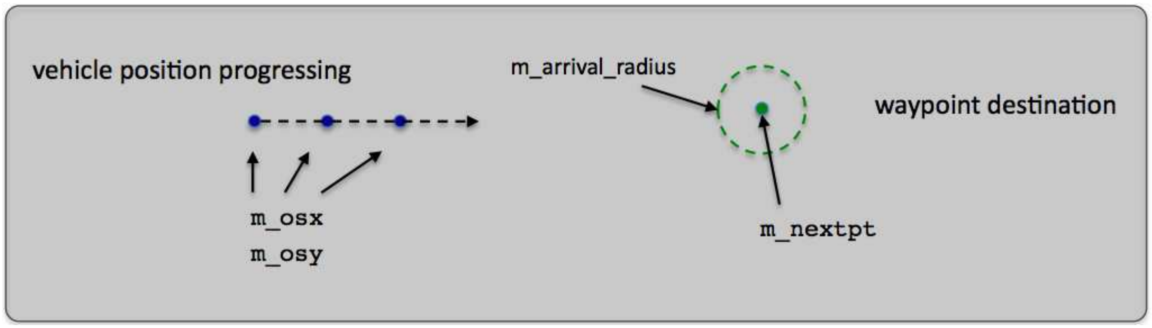


Figure 4: **The SimpleWaypoint behavior**: The SimpleWaypoint behavior works with a single waypoint. The location of the waypoint is stored in the local variable `m_nextpt` and is set during behavior configuration. The local variables `m_osx` and `m_osy` reflect the current vehicle (ownship) position updated at every helm iteration. The `m_arrival_radius` determines how close the vehicle needs to be from the waypoint destination before declaring completion.

The `BHV_SimpleWaypoint` class definition is given below in Listing 8. Note that it is declared to be a subclass of the `IvPBehavior` superclass on line 8. The three helm-invoked overloadable functions are declared on lines 13-15. The constructor is defined to take an `IvPDomain` as an argument. The helm will instantiate each behavior with the same helm-configured domain as an argument to a behavior constructor.

Listing 8 - BHV_SimpleWaypoint.h - the class definition for the "simple waypoint" behavior.

```
 1  #ifndef BHV_SIMPLE_WAYPOINT_HEADER
 2  #define BHV_SIMPLE_WAYPOINT_HEADER
 3
 4  #include <string>
 5  #include "IvPBehavior.h"
 6  #include "XYPoint.h"
 7
 8  class BHV_SimpleWaypoint : public IvPBehavior {
```

```
 9  public:
10    BHV_SimpleWaypoint(IvPDomain);
11    ~BHV_SimpleWaypoint() {};
12
13    bool         setParam(std::string, std::string);
14    void         onIdleState();
15    IvPFunction* onRunState();
16
17  protected:
18    void         postViewPoint(bool viewable=true);
19    IvPFunction* buildFunctionWithZAIC();
20    IvPFunction* buildFunctionWithReflector();
21
22  protected: // Configuration parameters
23    double       m_arrival_radius;
24    XYPoint      m_nextpt;
25    double       m_desired_speed;
26    std::string  m_ipf_type;
27
28  protected: // State variables
29    double   m_osx;
30    double   m_osy;
31  };
32
33  extern "C" {
34    IvPBehavior * createBehavior(std::string name, IvPDomain domain)
35    {return new BHV_SimpleWaypoint(domain);}
36  }
37  #endif
```

The two configuration parameters depicted in Figure 4, the waypoint and arrival radius, are declared on lines 23-124. The two remaining configuration parameters, `"speed"` and `"ipf_type"` are on the following two lines. The former sets the ideal speed for waypoint transiting, and the latter indicates the type of IvP function to be generated. Two different ways of generating an IvP function are implemented in this behavior to demonstrate two different tools. The last part, lines 33-36 are the hooks needed for each behavior class to implement the dynamic loading of behaviors into the helm. These lines are therefore not present for behaviors compiled into the IvP helm. These lines are very pertinent to the discussion of "extending" the helm.

5.2 The SimpleWaypoint Behavior Class Implementation

The class implementation is given in Listings 9-14 below.

5.2.1 The SimpleWaypoint Behavior Constructor

The first part contains the class constructor in lines 17-34. On line 18, a call to the base-class constructor is made with the given domain. A default for the behavior name is also set on line 20. On line 21, the behavior declares that the domain over which it will produce an IvP function is comprised of both the `course` and `speed` variables. If the domain given to the behavior by the helm in the constructor does not have either of these variables, a null IvP domain will result in line 21. A null domain will make the behavior thereafter not capable of running, and is considered a fatal error, prompting the helm to post all-stop output values. This is purposely drastic. Configuring the behaviors in a vehicle mission where one of the behaviors is not runnable is worthy of stopping the helm and addressing the problem. Since this condition is checked for on all behaviors on each helm iteration, this problem would always reveal itself at launch time, never during a mission, regardless of any dynamic behavior configurations during a mission.

On lines 25-27, default values for class member variables representing key behavior parameters are set in the constructor. In lines 30-31, class member variables representing behavior state variables are initialized. The grouping of member variables into two sets, one that represent parameter configurations and the other that otherwise represent behavior state maintained during operation, is merely a convention that has provided clarity in practice.

Listing 9 - BHV_SimpleWaypoint.cpp - The SimpleWaypoint Behavior Constructor.

```
 1  #include <cstdlib>
 2  #include <math.h>
 3  #include "BHV_SimpleWaypoint.h"
 4  #include "MBUtils.h"
 5  #include "AngleUtils.h"
 6  #include "BuildUtils.h"
 7  #include "ZAIC_PEAK.h"
 8  #include "OF_Coupler.h"
 9  #include "OF_Reflector.h"
10  #include "AOF_SimpleWaypoint.h"
11
12  using namespace std;
13
14  //-----------------------------------------------------------
15  // Procedure: Constructor
16
17  BHV_SimpleWaypoint::BHV_SimpleWaypoint(IvPDomain gdomain) :
18    IvPBehavior(gdomain)
19  {
20    IvPBehavior::setParam("name", "simple_waypoint");
21    m_domain = subDomain(m_domain, "course,speed");
22
23    // All distances are in meters, all speed in meters per second
24    // Default values for configuration parameters
25    m_desired_speed  = 0;
26    m_arrival_radius = 10;
27    m_ipf_type       = "zaic";
28
29    // Default values for behavior state variables
30    m_osx  = 0;
31    m_osy  = 0;
32
33    addInfoVars("NAV_X, NAV_Y");
34  }
35
```

Finally, on line 33, the behavior declares two variables, `NAV_X` and `NAV_Y`, representing vehicle ownship position. The IvP helm, containing this behavior, will need to register for these to variables on the behavior's behalf. This is the hook where the behavior tells the helm what it needs from the MOOSDB. It is from these two variables that the behavior will populate its variables `m_osx` and `m_osy` representing the current vehicle position.

5.2.2 The SimpleWaypoint Behavior `setParam()` Function

In Listing 10 below, a key overloadable behavior function is implemented, the `setParam()` function, in lines 39-69. This function handles the configuration of the behavior for its five parameters, "ptx", "pty", "speed", "radius", and "ipf_type". An example configuration for this behavior is given in Listing 15. Behavior parameters defined at the IvPBehavior superclass level, such as `name`, `condition`, `endflag`, etc., are handled in the `setParam()` function of the superclass. The helm, when it handles a behavior parameter from a `*.bhv` file, first attempts to handle the parameter at the superclass level. If the `IvPBehavior::setParam()` function returns `false`, the helm passes the parameter-value pair to the behavior's locally implemented version of `setParam()`.

Listing 10 - BHV_SimpleWaypoint.cpp - The `setParam()` *function.*

```
36  //---------------------------------------------------------------
37  // Procedure: setParam - handle behavior configuration parameters
38
39  bool BHV_SimpleWaypoint::setParam(string param, string val)
40  {
41    // Convert the parameter to lower case for more general matching
42    param = tolower(param);
43
44    double double_val = atof(val.c_str());
45    if((param == "ptx")  && (isNumber(val))) {
46      m_nextpt.set_vx(double_val);
47      return(true);
48    }
49    else if((param == "pty") && (isNumber(val))) {
50      m_nextpt.set_vy(double_val);
51      return(true);
52    }
53    else if((param == "speed") && (double_val > 0) && (isNumber(val))) {
54      m_desired_speed = double_val;
55      return(true);
56    }
57    else if((param == "radius") && (double_val > 0) && (isNumber(val))) {
58      m_arrival_radius = double_val;
59      return(true);
60    }
61    else if(param == "ipf_type") {
62      val = tolower(val);
63      if((val == "zaic") || (val == "reflector")) {
64        m_ipf_type = val;
65        return(true);
66      }
67    }
68    return(false);
69  }
70
```

A fair amount of error checking is done for parameter. For example, in setting the "speed" parameter, the string value is checked to ensure that is both numerical and larger than zero. Solid error checking implemented in this function is a very good idea that will save headaches down the road. This function should only return `true` if it has been passed a proper parameter-value pair. Another common practice is to perform a case insensitive parameter match, e.g., "pty" and "PTY" are both allowable configurations. This is done by converting the string representing the parameter to lower case in line 42. In this case, the `tolower()` function is defined in a local utility toolbox.

5.2.3 The SimpleWaypoint `onIdleState()` and `postViewPoint()` Functions

The `onIdleState()` function, lines 74-77, is only executed when the behavior is in the *idle* state, i.e., not in the *running* state. See Sections 4.4 and 4.6 for more on behavior states. In this behavior, the only task executed in the `onIdleState()` function is to publish a waypoint marker in the form of the MOOS variable `VIEW_POINT`.

Listing 11 - BHV_SimpleWaypoint.cpp - The `onIdleState()` *and* `postViewPoint()` *functions.*

```
71  //-----------------------------------------------------------
72  // Procedure: onIdleState
73
74  void BHV_SimpleWaypoint::onIdleState()
75  {
76    postViewPoint(false);
77  }
78
79  //-----------------------------------------------------------
80  // Procedure: postViewPoint
81
82  void BHV_SimpleWaypoint::postViewPoint(bool viewable)
83  {
84    m_nextpt.set_label(m_us_name + "'s next waypoint");
85    m_nextpt.set_type("waypoint");
86    m_nextpt.set_source(m_descriptor);
87
88    string point_spec;
89    if(viewable)
90      point_spec = m_nextpt.get_spec("active=true");
91    else
92      point_spec = m_nextpt.get_spec("active=false");
93    postMessage("VIEW_POINT", point_spec);
94  }
95
```

An example produced by this would be:

```
VIEW_POINT = "active,false:label,alder's next waypoint:type,waypoint:source,waypt_return:0,0,0"
```

In this case, due to the `"active,false"` component, the posting of this variable would serve to "erase" similar postings to this variable made in the `onRunState()` function described next. For more on how `VIEW_POINT` is consumed, see the documentation on the `pMarineViewer` application in [3].

5.2.4 The SimpleWaypoint Behavior `onRunState()` Function

Implementation of the `onRunState()` function is where the primary unique operation of the behavior is implemented. For the SimpleWaypoint behavior, the full function is in Listing 12 below. It is implemented in four parts:

- Part 1: Get the vehicle position from the information buffer.
- Part 2: Determine if the waypoint has been reached and possibly enter *complete* mode.
- Part 3: Build a status message regarding the waypoint for third party viewers.

- Part 4: Build an IvP function with either the ZAIC or Reflector tool.

In the first part, lines 101-109, information from the information buffer is retrieved regarding the vehicle's own position. This is done with the `getBufferDoubleVal()` function described in Section 4.5.1. In this behavior the result of the query to the buffer is stored in the `ok1` and `ok2` variables and subsequently checked and handled in lines 106-109. In this behavior, if essential information like the vehicle's own position is missing, a warning is posted (line 107) and the `onRunState()` function returns without producing an objective function (line 108). In such a case the behavior would be considered to be in the running state, but not the active state for the present iteration.

A fair point to raise regarding Part 1 is the possibility that the vehicle's position information is in the buffer but has become so old that it no longer reflects the vehicle's true current position. In other words, what if the navigation module on board the vehicle has somehow shut down? First, in most situations with a vehicle implementing the backseat/front-seat driver architecture described in [5] and [6], a heartbeat monitor for the navigation system is typically put in place at the larger autonomy system level and an all-stop would be invoked overriding the helm. However, for the sake of having some fail-safe redundancy *within* the helm to handle this situation, the `NO_STARVE` parameter could be used (Section 4.2.1) for this behavior, or any behavior since it is defined at the `IvPBehavior` superclass level. An example of it's usage is shown in Listing 15, setting a `NO_STARVE` threshold of 3 seconds for `NAV_X` and `NAV_Y`.

Listing 12 - BHV_SimpleWaypoint.cpp - The `onRunState()` *implementation.*

```
 96  //-----------------------------------------------------------
 97  // Procedure: onRunState
 98
 99  IvPFunction *BHV_SimpleWaypoint::onRunState()
100  {
101    // Part 1: Get vehicle position from InfoBuffer and post a
102    // warning if problem is encountered
103    bool ok1, ok2;
104    m_osx = getBufferDoubleVal("NAV_X", ok1);
105    m_osy = getBufferDoubleVal("NAV_Y", ok2);
106    if(!ok1 || !ok2) {
107      postWMessage("No ownship X/Y info in info_buffer.");
108      return(0);
109    }
110
111    // Part 2: Determine if the vehicle has reached the destination
112    // point and if so, declare completion.
113    double dist = hypot((m_nextpt.x()-m_osx), (m_nextpt.y()-m_osy));
114    if(dist <= m_arrival_radius) {
115      setComplete();
116      postViewPoint(false);
117      return(0);
118    }
119
120    // Part 3: Post the waypoint as a string for consumption by
121    // a viewer application.
122    postViewPoint(true);
123
124    // Part 4: Build the IvP function with either the ZAIC tool
125    // or the Reflector tool.
126    IvPFunction *ipf = 0;
127    if(m_ipf_type == "zaic")
128       ipf = buildFunctionWithZAIC();
129     else
130       ipf = buildFunctionWithReflector();
131     if(ipf == 0)
132       postWMessage("Problem Creating the IvP Function");
133
134     return(ipf);
135   }
136
```

In Part 2 of the `onRunState()` function, in lines 111-118 of Listing 12, the determination of waypoint arrival is made. This is just a simple comparison between the current distance of the vehicle and the waypoint to the configured arrival radius in the parameter `m_arrival_radius`. If a determination of arrival is made, the behavior calls the `setComplete()` function. This function is defined at the behavior superclass level and was described in detail in Section 4.5.1. The invocation of this function will put the behavior in the group of *completed* behaviors on the next helm iteration. In the current iteration this behavior would be considered in the *running* state, but not the *active* state since the `onRunState()` function returns (line 117) without generating an objective function. See [6] for more on behavior run states.

In Part 3, the behavior generates a visual artifact for consumption by a viewer, for rendering the waypoint the behavior is using as its destination. This point is shown in Figures 9 and 10 with the label `"Alder's next waypoint"`. An example produced by this would be:

```
VIEW_POINT = "label=alder's next waypoint,type=waypoint source=transit,x=60,y=-40"
```

Compare this to the value for `VIEW_POINT` generated in the `onIdleState()` function described in Section 5.2.3. This variable-value pair is generated by the behavior for posting on each invocation of the `onRunState()` even though the value posted does not generally change between iterations. The posting of the variable-value pair is done with the `postMessage()` function, described in Section 4.5.1. The invocation of `postMessage()` will result in an actual post to the MOOSDB only if the value of string posted changes. Successive duplicate postings are filtered out by the Duplication Filter. The Duplication Filter is described in detail in [6]. The `postMessage()` function was discussed in Section 4.5.1 on page 36.

In Part 4 of the `onRunState()` function, in lines 124-135 of Listing 12, an IvP function is generated over a domain of heading and speed choices to reflect the goal of reaching a waypoint given a current vehicle position. For the purposes of providing an example usage of the IvPBuild Toolbox, the behavior is implemented to produced an IvP function using two different methods of the toolbox. The SimpleWaypoint behavior can be configured with the `"ipf_type"` parameter, as shown in Listing 10, to accept either the configuration of `"zaic"` or `"reflector"`. In the example mission, mission "Alder" described below in Section 5.3, the helm is configured in Listing 15 to use the ZAIC tool on the outbound transit trip, and the Reflector tool on the return trip. The implementation of the `onRunState()` function merely checks the type of IvP function desired and makes the appropriate call to either the `buildFunctionWithZAIC()` function or the `buildFunctionWithReflector()` function. These two functions are described next.

5.2.5 The SimpleWaypoint Behavior `buildFunctionWithZAIC()` Function

When the SimpleWaypoint behavior is configured to generate an IvP function with the ZAIC tool, it invokes the function `buildFunctionWithZAIC()`, shown below in Listing 13. Of the three ZAIC tools described in Section 7, the `ZAIC_PEAK` is used in this behavior. It is used to generate two one-variable IvP functions. The first function is defined over the `speed` decision variable in lines 142-151. The second function is defined over `heading` decision variable. The functions are shown in Figure 5.

Listing 13 - BHV_SimpleWaypoint.cpp - The `buildFunctionWithZAIC()` *implementation.*

```
137 //-----------------------------------------------------------
138 // Procedure: buildFunctionWithZAIC
139
140 IvPFunction *BHV_SimpleWaypoint::buildFunctionWithZAIC()
141 {
142   ZAIC_PEAK spd_zaic(m_domain, "speed");
143   spd_zaic.setSummit(m_desired_speed);
144   spd_zaic.setPeakWidth(0.5);
145   spd_zaic.setBaseWidth(1.0);
146   spd_zaic.setSummitDelta(0.8);
147   if(spd_zaic.stateOK() == false) {
148     string warnings = "Speed ZAIC problems " + spd_zaic.getWarnings();
149     postWMessage(warnings);
150     return(0);
151   }
152
153   double rel_ang_to_wpt = relAng(m_osx, m_osy, m_nextpt.x(), m_nextpt.y());
154   ZAIC_PEAK crs_zaic(m_domain, "course");
155   crs_zaic.setSummit(rel_ang_to_wpt);
156   crs_zaic.setPeakWidth(0);
157   crs_zaic.setBaseWidth(180.0);
158   crs_zaic.setSummitDelta(0);
159   crs_zaic.setValueWrap(true);
160   if(crs_zaic.stateOK() == false) {
161     string warnings = "Course ZAIC problems " + crs_zaic.getWarnings();
162     postWMessage(warnings);
163     return(0);
164   }
165
166   IvPFunction *spd_ipf = spd_zaic.extractIvPFunction();
167   IvPFunction *crs_ipf = crs_zaic.extractIvPFunction();
168
169   OF_Coupler coupler;
170   IvPFunction *ivp_function = coupler.couple(crs_ipf, spd_ipf, 50, 50);
171
172   return(ivp_function);
173 }
174
```

The first step in creating an `IvPFunction` with the `ZAIC_PEAK` tool is to create an instance of the `ZAIC_PEAK`, on line 142, passing it the `IvPDomain` used by the behavior and set in the behavior constructor in Listing 9. The ZAIC constructor is also passed the name of the one variable in the `IvPDomain` for which to create the `IvPFunction`. The `ZAIC_PEAK` parameters, set in lines 143-146, are described in detail in Section 7.1.1. In lines 147-151, a check is made to determine whether the `ZAIC_PEAK` instance has been configured properly. The `stateOK()` function returns `false` if there were any configuration problems, and the string returned by `getWarnings()` function on line 148 will provide insight into any configuration errors. The `postWMessage()` function on lines 149 and 162 will result in the helm posting to the MOOSDB the variable `BHV_WARNING` with the contents of string `warnings`. The `postWMessage()` function is discussed in Section 4.5.1 on page 37.

The second one-variable function, defined over `course`, is created with a second `ZAIC_PEAK` instance in lines 153-164. First, the angle between the current vehicle position and the waypoint destination is calculated on line 153, with a call to the `relAng()` function, defined in one of the MOOS-IvP utility libraries. The creation and configuration of the `ZAIC_PEAK` instance proceeds much in same way as for the first one. In this case, the `valuewrap` parameter is set to `true` on line 159 to indicate that the domain values should "wrap around", that is, a course of 350 is 20 degrees separated from a course of 10 degrees, not separated by 340 degrees. The `valuewrap` parameter is discussed in Section 7.1.3 on page 65.

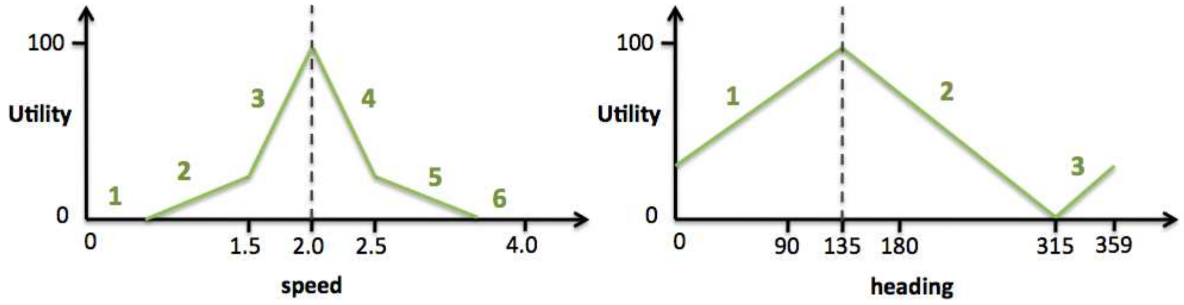


Figure 5: **IvP functions produced by the ZAIC_PEAK Tool**: The two functions produced by the SimpleWaypoint behavior by use of the `ZAIC_PEAK` tool would produce a function over `speed` with six pieces and a function over `heading` with three pieces.

When these two functions are coupled using the `Coupler` tool, lines 169-170, an IvP function over the coupled decision 2D space is created as shown in Figure 6 below. The Coupler tool is discussed in Section 6.2.3 on page 61.

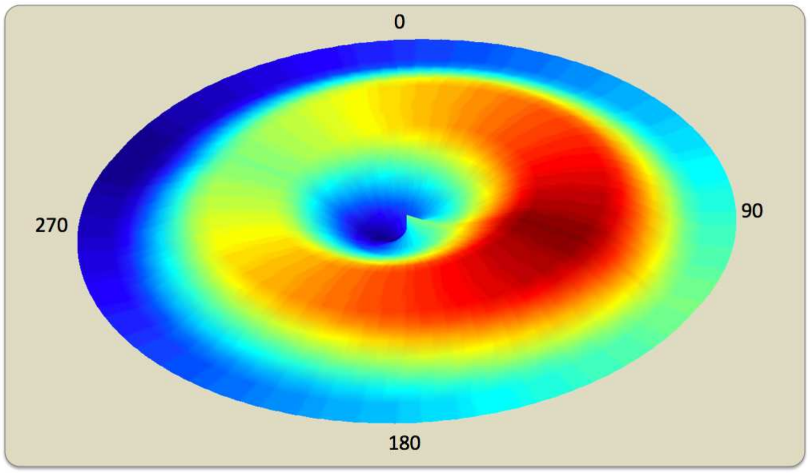


Figure 6: **An IvP function coupled from two one-variable functions**: This function is created by coupling the pair of one-variable functions of Figure 5 using the `OF_Coupler` tool. Decisions of increasing speed are represented by points on the function radially farther from the center.

The two one-variable functions are combined with an equal weight of 50 on line 170. The choice of relative weight has a distinct influence over the resulting function. In Figure 7 below, two IvP functions similarly generated, but with alternative weights are shown. The Coupler, by default, normalizes the combined function to the range of $[0, 100]$. This can be turned off, or normalized with a different range as described in Section 6.2.3. The range of $[0, 100]$ is a common range for functions returned by an IvP behavior to the IvP Solver.

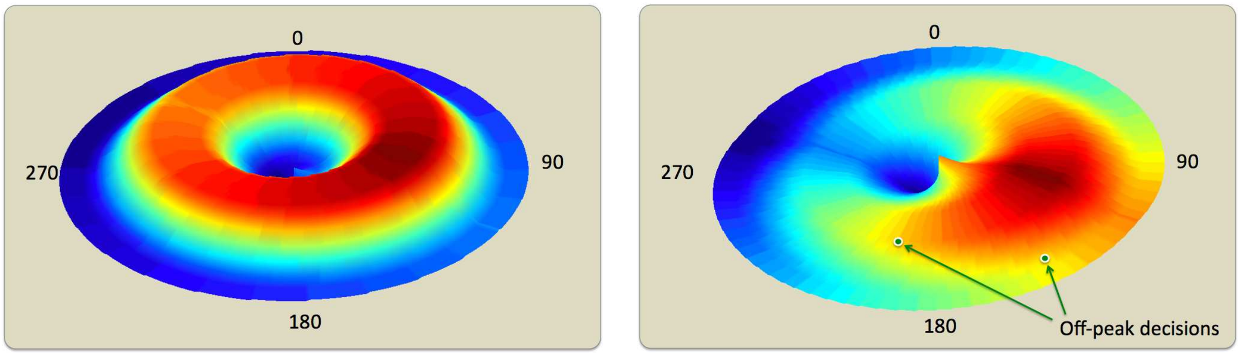


Figure 7: **Two IvP functions coupled from the same two one-variable functions**: These functions were created by coupling the pair of one-variable functions of Figure 5 using the `OF_Coupler` tool. They differ only in the relative weight applied to each function. The one on the left had a weight of 75 to the `speed` function and a weight of 25 to the `course` function. These weights are reversed on the right. The function on the right shows two off-peak decision points with equal utility rating. Decisions of increasing speed are represented by points on the function radially farther from the center.

In each IvP function in Figures 6 and 7, the location of the *peak* of the function is the same. When this behavior is the only active behavior, the path taken by the vehicle will be the same regardless of the weights chosen to combine the two one-variable functions with the Coupler. The only thing that matters is the value of the `summit` parameters passed to the two ZAIC tools creating the two one-variable functions. The off-peak characteristics of the function begin to matter when the behavior is coordinated with functions from other behaviors. In the function on the right in Figure 7, two off-peak decisions with equal utility values are shown. One point represents a decision with the heading more toward the destination with a speed much higher than the desired speed, and the other decision represents a heading less toward the destination but with speed near the optimal speed. In the absence of mission metrics that clarify the relative utility of sub-optimal transit paths versus sub-optimal speeds, the construction of the off-peak shape of the objective function is typically a subjective decision of the behavior implementor.

5.2.6 The SimpleWaypoint Behavior `buildFunctionWithReflector()` Function

The Reflector tool can be used to generate objective functions that cannot otherwise be formed as the product of the coupling of two independent functions. This gives the behavior implementor more freedom to generate functions with off-peak characteristics more in-line with the goals of the behavior. The Reflector tool is described in detail in Sections 8 and 9.

The use of the Reflector tool in the SimpleWaypoint behavior is given in Listing 14 below. The Reflector generates an IvP function approximation of an underlying function, an instance of the `AOF_SimpleWaypoint` class. The underlying function and the IvP function are defined over the same domain. This domain is passed to the underlying function in its constructor (line 183). The underlying function is passed required parameters (lines 184-188) and initialized (line 189). If any part of the initialization fails, a null IvP function is returned (line 180, 196).

Listing 14 - BHV_SimpleWaypoint.cpp - The `buildFunctionWithReflector()` *implementation.*

```
175  //----------------------------------------------------------
176  // Procedure: buildFunctionWithReflector
177
178  IvPFunction *BHV_SimpleWaypoint::buildFunctionWithReflector()
179  {
180    IvPFunction *ivp_function = 0;
181
182    bool ok = true;
183    AOF_SimpleWaypoint aof_wpt(m_domain);
184    ok = ok && aof_wpt.setParam("desired_speed", m_desired_speed);
185    ok = ok && aof_wpt.setParam("osx", m_osx);
186    ok = ok && aof_wpt.setParam("osy", m_osy);
187    ok = ok && aof_wpt.setParam("ptx", m_nextpt.x());
188    ok = ok && aof_wpt.setParam("pty", m_nextpt.y());
189    ok = ok && aof_wpt.initialize();
190    if(ok) {
191      OF_Reflector reflector(&aof_wpt);
192      reflector.create(1000);
193      ivp_function = reflector.extractIvPFunction();
194    }
195
196    return(ivp_function);
197  }
```

The Reflector tool does its work (lines 191-193) after it has been determined that a proper instance of the underlying function, `AOF_SimpleWaypoint`, has been created and initialized. The

Reflector tool has several options for creating a piecewise defined IvP function, described later in Sections 8 and 9. The simplest method is to specify the number of pieces desired in the piecewise function, in this case 1000 pieces were requested, on line 192. After creation, the IvP function is extracted from the Reflector (line 193), and returned (line 196).

An example of the IvP function created with the Reflector is shown in Figure 8. The function represents a preference for maneuvers that bring the vehicle toward the waypoint at a desired speed. The values of off-peak areas are evaluated based on (a) the rate of closure, (b) the rate of detour, and (c) the deviation from the desired speed. For this function generated by the Reflector, and the particular underlying function, it is not possible to generate a function of equal form using the ZAIC tools.

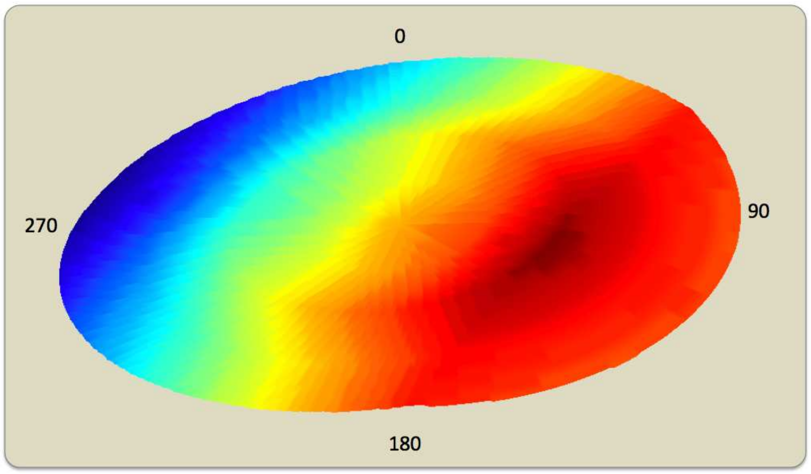


Figure 8: **An IvP function created with the Reflector tool**: The function represents a preference for maneuvers that bring the vehicle toward the waypoint at a desired speed. The values of off-peak areas are evaluated based on (a) the rate of closure, (b) the rate of detour, and (c) the deviation from the desired speed. Decisions of increasing speed are represented by points on the function radially farther from the center.

5.3 Running an Example Mission with the SimpleWaypoint Behavior

An example mission file, `alder.moos`, and behavior file, `alder.bhv`, have been configured to demonstrate the usage of the SimpleWaypoint behavior. The files may be found in the `moos-ivp-extend` tree. Refer back to Section 2.2 for information on obtaining this tree from the web. The example mission with the SimpleWaypoint behavior is called the Alder mission and is found by:

```
> cd moos-ivp-extend/missions/alder
> ls
> README   alder.bhv   alder.moos
```

The behavior file is given in Listing 15 below. The SimpleWaypoint behavior is used twice. It is used to transit to a point and then to return to the starting point. The transiting use of the behavior is configured in lines 7-20, and the returning use of the behavior in lines 23-35. See [6] for further information regarding behavior files and their usage.

Listing 15 - The `alder.bhv` *file - For running the Alder example mission.*

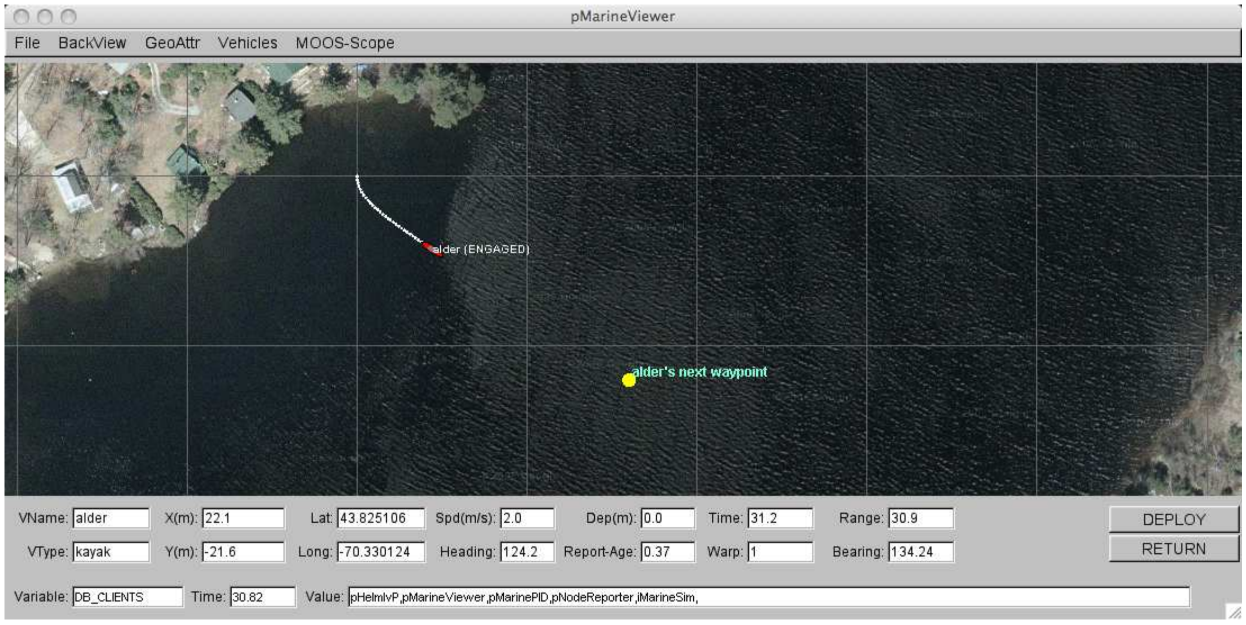
```
//--------    FILE: alder.bhv   -------------

initialize   DEPLOY = false
initialize   RETURN = false

//-----------------------------------------------
Behavior = BHV_SimpleWaypoint
{
  name       = transit_to_point
  pwt        = 100
  condition  = RETURN = false
  condition  = DEPLOY = true
  endflag    = RETURN = true

       speed = 2.0   // meters per second
      radius = 8.0
         ptx = 60
         pty = -40
    ipf_type = zaic
}

//-----------------------------------------------
Behavior = BHV_SimpleWaypoint
{
  name       = waypt_return
  pwt        = 100
  condition  = RETURN = true
  condition  = DEPLOY = true

       speed = 2.0
      radius = 8.0
         ptx = 0
         pty = 0
    ipf_type = reflector
}
```

Both behaviors are idle upon startup. Presumably the transit behavior is activated first by setting `DEPLOY=true`, and the second instance of the behavior is activated when the transit behavior completes and sets its endflag. The `alder.moos` file is not discussed here, but may be examined in the tree.

The example mission may be started by:

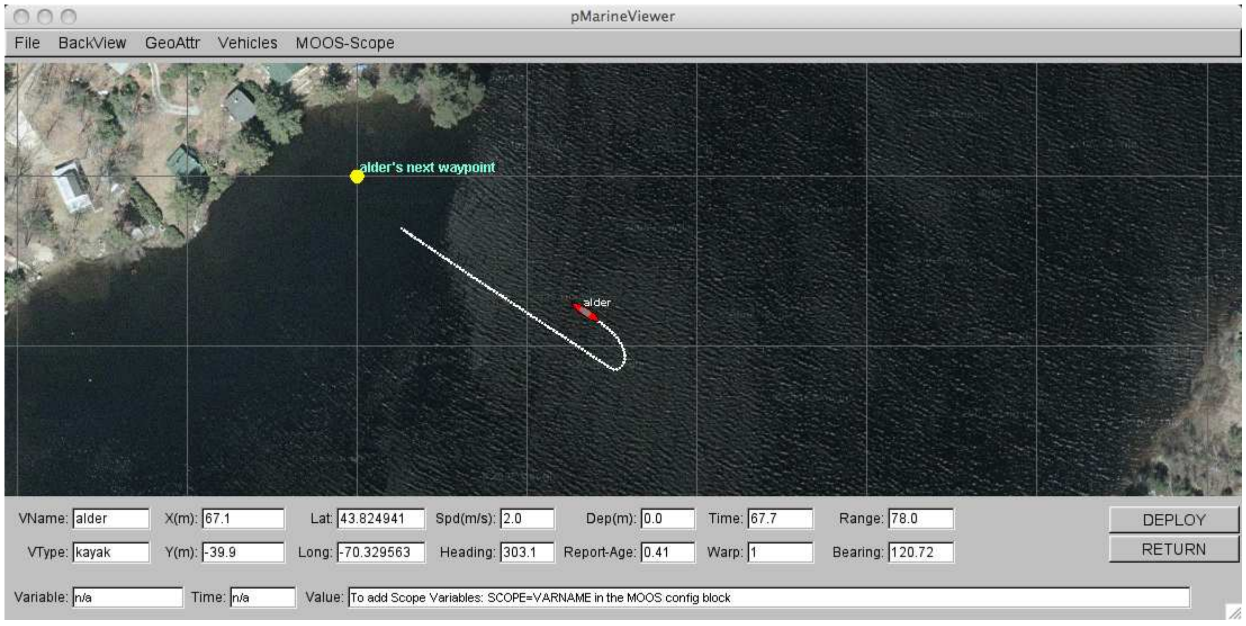
```
> cd moos-ivp-extend/missions/alder
> pAntler alder.moos
```

The `pMarineViewer` window should launch, and look similar to image in Figure 9. After clicking on the `DEPLOY` button in the lower right corner, the transiting instance of the SimpleWaypoint behavior becomes active and the vehicle begins to form a track to the waypoint as shown. Clicking on the `DEPLOY` button initiates a MOOS poke on the MOOSDB connected to both the `pMarineViewer`, and `pHelmIvP` application. This mission setup is quite similar to the Alpha mission discussed in [6].



VName: alder X(m): 22.1 Lat: 43.825106 Spd(m/s): 2.0 Dep(m): 0.0 Time: 31.2 Range: 30.9
VType: kayak Y(m): -21.6 Long: -70.330124 Heading: 124.2 Report-Age: 0.37 Warp: 1 Bearing: 134.24
Variable: DB_CLIENTS Time: 30.82 Value: pHelmIvP,pMarineViewer,pMarinePID,pNodeReporter,iMarineSim,

Figure 9: **The SimpleWaypoint behavior in action**: After the user clicks on the `DEPLOY` button, the condition for the transiting SimpleWaypoint behavior is satisfied (line 13 in Listing 15).



VName: alder X(m): 67.1 Lat: 43.824941 Spd(m/s): 2.0 Dep(m): 0.0 Time: 67.7 Range: 78.0
VType: kayak Y(m): -39.9 Long: -70.329563 Heading: 303.1 Report-Age: 0.41 Warp: 1 Bearing: 120.72
Variable: n/a Time: n/a Value: To add Scope Variables: SCOPE=VARNAME in the MOOS config block

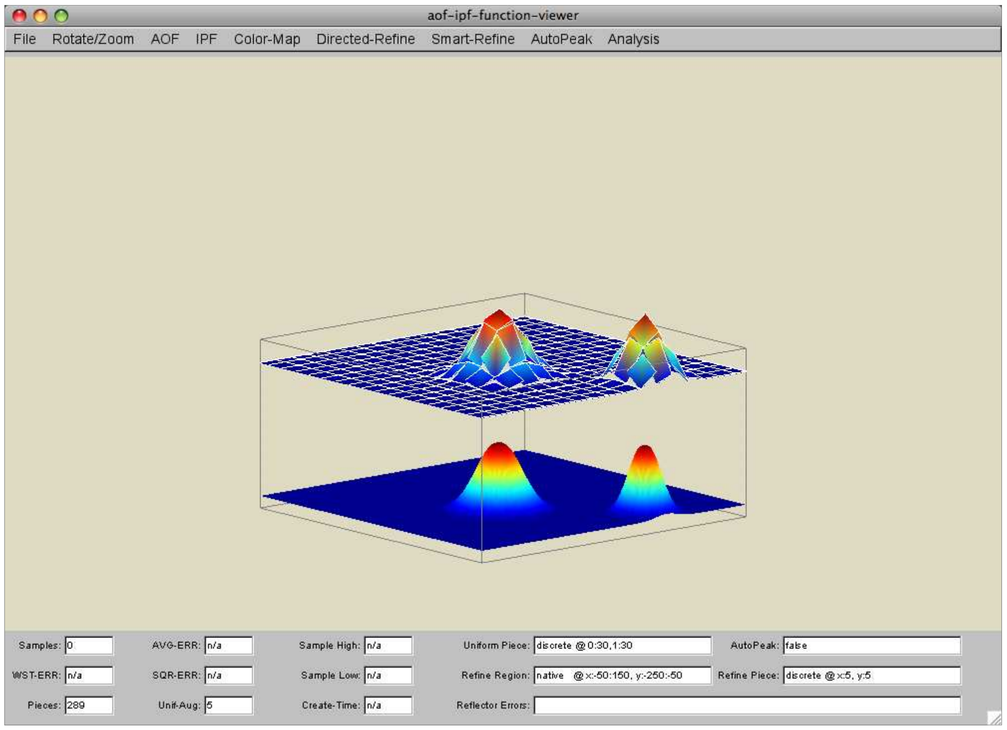
Figure 10: **The SimpleWaypoint behavior in action**: After the vehicle has reached the waypoint prescribed in the transiting instance of the SimpleWaypoint behavior, the second instance of the SimpleWaypoint behavior, returning to the start point, becomes active.

6 Introduction to the IvPBuild Toolbox

The IvPBuild Toolbox is a set of C++ classes and algorithms for building IvP functions. The primary objective is to provide tools to the implementors of new helm behaviors that are fast and easy to use. In the behavior implementation example in Listing 11 on page 45, the creation of an IvP function required only about a dozen lines of code using two different methods available in the IvPBuild Toolbox.

6.1 Brief Overview

An instance of the class `IvPFunction` is the primary output of a behavior on each helm iteration, and is comprised of anywhere from a handful to thousands of "pieces" that approximate the utility function of the behavior. An example IvP function approximating a utility function is shown below in Figure 11.



```
Samples: 0        AVG-ERR: n/a    Sample High: n/a    Uniform Piece: discrete @0:30,1:30                AutoPeak: false
WST-ERR: n/a      SQR-ERR: n/a    Sample Low: n/a     Refine Region: native   @x:-50:150, y:-250:-50    Refine Piece: discrete @x:5, y:5
Pieces: 289       Unif-Aug: 5     Create-Time: n/a    Reflector Errors:
```

Figure 11: **An IvP function approximating an underlying function**: The `fview` tool is used to render an IvP function with 289 pieces to approximate a given function, shown below the IvP function.

The toolbox contains tools for making simple one-variable objective functions (the "ZAIC" tools) as well as functions over N variables (the "Reflector" tools). The primary contribution of the user (behavior implementor) is to provide the underlying utility function provided to the toolbox. The IvP function approximation is generated automatically given user parameter preferences.

6.1.1 Where to Get the IvPBuild Toolbox

The IvPBuild Toolbox is part of the standard moos-ivp bundle distributed from www.moos-ivp.org. See Section 1.4. In the software tree it is entirely contained in the module `lib_ivpbuild`.

6.1.2 What is an Objective Function?

An objective function is a function like any other, a mapping from a domain to a range. In the case where the domain variables correspond to decisions or choices, and the range corresponds to the utility with respect to a particular user objective, the function is often called an "objective" function, or "utility" function.

6.1.3 What is Multi-objective Optimization?

The term multi-objective optimization refers to a situation where there are multiple objective functions defined over the same domain, i.e., decision space, and the ideal goal is to find a point in the decision space that optimizes all functions simultaneously. Rarely is such a mutually agreeable decision available and typically the functions can be said to be "competing". Techniques vary widely on how to handle this. A simple technique would be to rank order the functions and optimize the most important first, and so on. Another technique involves setting a competence threshold for each and choosing from decisions that satisfy a minimum competence for each function. For an in-depth treatment see [9], [10], [11], [15], [16], [17], [18], [19].

Many techniques for optimization are predicated on there being a user involved in the decision process who can interactively alter parameters of the problem until an agreeable resolution emerges. In these cases the notion of Pareto optimality, [14], often plays a central role. A Pareto optimal solution is one that cannot be improved in regard to one objective unless it comes at the expense of another objective. Typical user-interactive multi-objective optimization techniques involve letting the user explore the Pareto frontier, i.e., those solutions that are all Pareto optimal differing only on the user's value function or relative preference in importance of objectives.

In repeatedly applying multi-objective optimization to the output of behaviors in an on-board *autonomous* decision making system, there is no user involved by definition. There is no exploration of the Pareto frontier since that exploration requires a user. Instead, part of the autonomy process involves also setting the value function, i.e., the relative importance of objectives. In the IvP helm, this value function is reflected by *priority weights* assigned to each function, and the multi-objective optimization problem is reduced to a single objective optimization problem, given k functions and w_i being the weight of the ith function:

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

The properties of IvP multi-objective optimization and solution algorithms are discussed in the section "The IvP Solver and Behavior Priority Weights" in [6].

6.1.4 What is an IvP Function?

An IvP function is a piecewise linearly defined function where each piece has an upper and lower boundary (or *interval*) on the decision space and linear function defined over the piece. An IvP function is defined over a domain that itself has an upper and lower boundary for each decision variable. Furthermore, the domain is comprised of equally spaced discrete points, and therefore each piece is defined over a finite set of points in the domain. An IvP function is typically an *approximation* of the user's underlying utility function. The fidelity of this approximation can be controlled by the user of the toolbox by deciding how many pieces are used in the approximation. Since the size or extents of each piece may vary within a function, the toolbox methods may also create functions that user smaller pieces where the underlying function is less amenable to local linear approximations.

An IvP function as an instance of the `IvPFunction` class defined as part of the `lib_ivpcore` module included in the basic software bundle distributed from `www.moos-ivp.org`.

6.1.5 Why the IvP Function Construct? A Brief Description of the Solver

The IvP function construct was chosen because it balances three aspects needed for use in the extendable behavior-based autonomy philosophy.

- Flexibility: a piecewise defined function approximation can be formed from any underlying function and thus the behavior author is not compelled to produce objective functions of a restricted form, such as convex or continuous functions. The behavior author is free to innovate. The behavior author typically has insight into the degree of fidelity needed to faithfully reflect the underlying utility function.
- Speed: the IvP function constructs, once produced, can be exploited by solution algorithms to give very fast solutions with a guarantee of global optimality modulo the error introduced in function approximation.
- Accuracy: a piecewise defined function can be highly accurate for a few reasons, (a) by controlling the piece size and distribution the approximation can be made to be as accurate as needed. (b) by being free to approximate any underlying function form, a piecewise function may better reflect a behavior's utility. (c) by allowing for guaranteed globally optimal solutions in the resulting optimization problem, errors of this type are eliminated.

The IvP Solver uses a branch-and-bound method to search through the combination space of pieces, one from each of k contributing function. Since each point in the decision space is contained in exactly one piece in each function, the optimal decision corresponds to a k-tuple of pieces. Thus finding the optimal k-tuple guarantees that the optimal point in the decision space has been found. A leaf node in the tree is simply the "intersection" of pieces from contributing functions. Likewise the intersection of interior functions at a leaf node is simply the sum of the linear functions of each contributing piece. Both the intersection of rectilinear pieces and the sum of linear functions be rapidly and simply computed. For a detailed description of the IvP solver solution algorithms, see [4].

6.1.6 Properties of the `IvPDomain` Class

The domain of an IvP function is an instance of the class `IvPDomain`. It is the same between all functions produced by all behaviors. The domain has a finite set of labeled variables with a lower and upper bound for each variable, and an integer number of evenly spaced points between the bounds. The domain is also referred to as the *decision space*. The 2D base of the cube in Figure 11 represents the domain. A point in the domain is contained in exactly one piece of an IvP function. The domain is built by the helm at the time of launch, and a copy is handed to each behavior in its constructor to ensure uniformity between behaviors. Listing 16 shows an example domain with three variables as it would be specified within the MOOS configuration block for the pHelmIvP process.

Listing 16 - An IvP domain with three variables, as specified in `pHelmIvP` *configuration.*

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

Each line augments the initially null domain with a new variable. The first of four arguments is the variable name, e.g., `course`. The second and third arguments indicate the lower and upper bound of the variable. They are integers here, but could be floating point values. The last of four arguments is the number of points in the domain for that variable. This domain would have $(360 * 16 * 101)$ 581,760 distinct possible decisions.

The behavior author, using the IvPBuild Toolbox, only needs to create a black-box function routine able to evaluate any point in the IvP domain with respect to the objectives of that behavior. To use the toolbox, this routine needs to reside within an implementation of a class that subclasses the `AOF` class described in Section 8.2. Although behaviors share a common domain, they can be defined over a different subset of variables as long as the common variables match in extents. For example, a behavior in a helm configured with the domain above could be defined over the following sub-domain:

```
Domain  = depth:0:500:101
```

It could not be defined over:

```
Domain  = depth:0:100:101
```

To facilitate the proper creation of sub-domains, the following function is provided in the build toolbox (in `BuildUtils.h` in `lib_ivpbuild`):

```
IvPDomain  subDomain(IvPDomain original_domain, string variable_names);
```

The first argument is the original domain. The string argument is a comma separated list of variable names to be included in the sub-domain. If a variable is named in the argument that doesn't exist in the original domain, an empty domain is returned. This is detected by checking the size of the domain with a call to `domain.size()`, which returns the number of domain variables. A proper sub-domain of the domain shown in Listing 16, with only the `depth` variable, could be created with the following function call:

```
#include "BuildUtils.h"
...
domain = subDomain(original_domain, "depth");
```

It is common for a behavior to declare its sub-domain in its constructor, even if it is expected to be the same as the total helm domain. See for example line 21 in Listing 9 on page 43. A call to `subDomain()` has no effect if it names all of the original variables. By declaring a sub-domain in the behavior's constructor, problems can be avoided later if the overall helm domain is expanded. If the function call above erroneously creates a null domain for the behavior, the helm detects this automatically in the `isRunnable()` function call described in Section 4.4. This will cause an error message to be posted to the `BHV_ERROR` variable and result in the helm posting all-stop values to its actuators.

6.2 Tools Available in the IvPBuild Toolbox

The IvPBuild Toolbox contains a few different sets of tools. (a) The ZAIC tool is used for creating IvP functions with only one decision variable. (b) The basic Reflector tool is used for creating IvP functions over N coupled variables. (c) The advanced Reflector tools are an extension of the basic Reflector tools that allow for piecewise defined functions with non-uniform pieces. (d) The Coupler tool allows a pair of decoupled IvP functions to be converted to a single coupled IvP function. (e) The encoding/decoding tools allow IvP functions to be converted to a string representation and vice versa.

6.2.1 The ZAIC Tools for Functions with One Variable

The ZAIC tools are used for functions defined over a single decision variable. An example is shown in Figure 12 where a fictional behavior may want to keep the vehicle in the so-called "deep sound channel" or "SOFOR (SOund Fixing and Ranging) channel". This is a horizontal layer in the ocean around which the speed of sound is at a minimum and where sound, especially at low frequencies, may travel for thousands of meters with little loss of signal (See [1], [2]).

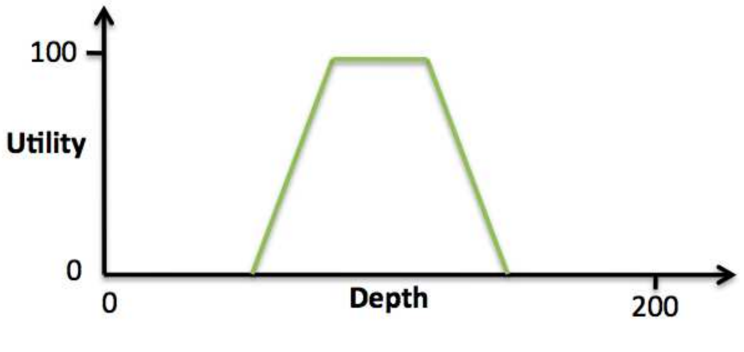


Figure 12: **An objective function with a single decision variable**: This function assigns a maximum utility to depths in a range of roughly 100 ± 20 meters. A linear decrease in utility is associated with depths outside this interval up to another additional 20 meters.

A piecewise defined IvP function can be constructed to represent this function using five pieces. Assuming the "depth" decision space is 0 to 200 meters at one meter increments, the intervals would be: $[0, 59], [60, 79], [80, 120], [121, 140], [141, 200]$. The linear function for each piece also needs to be set. This is not terribly difficult, but it is tedious and prone to human error. Instead, the `ZAIC_PEAK` utility is a tool in the ZAIC toolbox used for automating the production of IvP piecewise functions of the form shown in Figure 12. It is described in Section 7.

6.2.2 The Reflector Tool for Functions with Multiple Variables

The Reflector tool is used for creating IvP functions over n decision variables where $n > 2$. The Reflector was used to generate the IvP function rendered in Figure 11 on page 55. The tools do work for $n = 1$ but one variable functions are typically handled with the ZAIC tools described above. The Reflector produces an IvP function approximation of a given underlying function, where the underlying function is provided to the Reflector in the form of an instance of a class containing the underlying function implementation, and a specification of the IvP domain. The basic use of the Reflector is described in detail in Section 8, but the basic usage boils down to the following:

- Create an instance of the underlying function to be approximated.
- Create an instance of the Reflector, passing it the underlying function.
- Invoke the Reflector with a requested number of pieces.
- Retrieve the new IvP function from the Reflector.

The basic usage of the Reflector involves only the choosing the number of pieces used in the IvP function representation. By choosing only the number, pieces of uniform size will be used in the function. This suffices for most applications, but there are ways to produce a function that is both more accurate and uses less pieces by exploring advanced options and algorithms of the Reflector. These include:

- The *Directed Refinement* Reflector option.
- The *Smart Refinement* Reflector option.
- The *AutoPeak Refinement* Reflector option.

The details of these advanced options are discussed in Section 9. Each of the advanced tools are used after an initial basic uniform function has been generated. The *directed refinement* option allows the user to specify subsets of the domain and use pieces of different sizes for that region only. The *smart refinement* option asks the Reflector to estimate the fit of each piece as it is generated in terms of accuracy in approximating the underlying function and performs further refinement on those pieces that need it the most. The *autopeak refinement* option repeatedly refines the single piece containing the maxima of the underlying function until that point is contained in a piece containing only that point.

6.2.3 The Coupler Tool for Coupling Two Decoupled IvP Functions

Two IvP functions defined over different variables can be combined to form a single IvP function defined over the union of the two sets of variables. The basic usage of the Coupler can be summarized as follows:

- Create the two independent IvP functions.
- Create an instance of the `OF_Coupler` class.
- Pass the two functions to the Coupler.
- Retrieve the new IvP function from the Coupler.

This tool was used in the example SimpleWaypoint behavior of Section 5 in Listing 12 two couple two one-variable IvP functions. When a Coupler is passed the two IvP function pointers, it takes over ownership of the functions, i.e., it deletes the two one-variable functions when the Coupler object is deleted. When a coupled IvP function is extracted from the Coupler, ownship of the IvP function is passed to the caller, i.e., the caller is responsible for deleting the IvP function.

7 The ZAIC Tools for Building One-Variable IvP Functions

The ZAIC tools are part of the IvP Build Toolbox for facilitating the building of IvP functions over a single domain variable. There are three tools - ZAIC_PEAK, ZAIC_HEQ, ZAIC_LEQ. To use the tools, in short, one creates a instance of the corresponding class, sets some parameters, and then extracts an IvP function. The tools described in Section 8 for n-variable functions can also be used for building one-variable functions but are perhaps overkill for certain classes of common one-variable functions that motivated the ZAIC tools. The term ZAIC is not an acronym, but merely a play on the word mosaic.

7.1 The ZAIC_PEAK Tool

7.1.1 Brief Overview

The ZAIC_PEAK tool is designed with the objective function shown in Figure 13 in mind. There is a identifiable preferred single decision choice (the `summit`) with maximum utility, and then a gradual drop in utility as the variable value varies from the preferred choice.

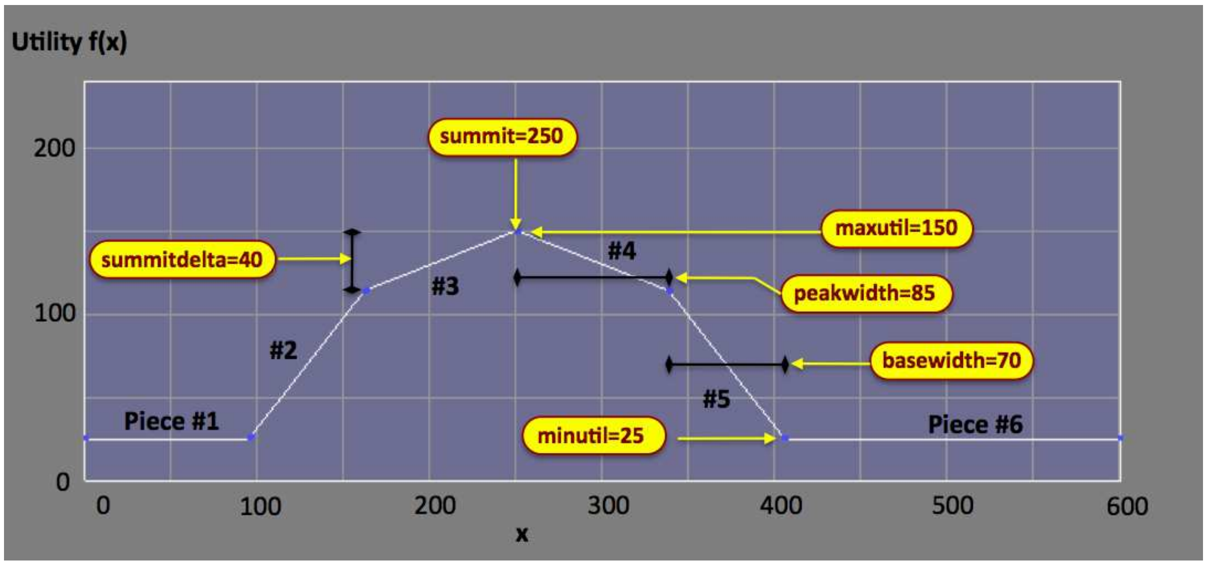


Figure 13: **The ZAIC_PEAK tool**: defines an IvP function over one variable defined by the six parameters shown here. In the case rendered here, the tool would create an IvP function with six pieces. The function rendered was created with `summit=180`, `peakwidth=85`, `basewidth=70`, `maxutil=150`, `minutil=25`, `summitdelta=40`.

The form in which the utility drops is dependent on the settings of the six parameters shown in the figure. The `summit`, `peakwidth`, and `basewidth` values are given in units native to the decision variable, while the `summitdelta`, `minutil`, and `maxutil` values are given in terms of units of utility.

7.1.2 The ZAIC_PEAK Parameters and Function Form

The ZAIC_PEAK tool accepts six parameters in defining $f(x)$. The `summit` parameter is the point of maximum utility. The `minutil` parameter is the minimum value of $f(x)$, with a default value of zero.

The `maxutil` is the maximum value of $f(x)$, with a default value of 100. The utility of the function drops off linearly in two stages. In the first stage the utility drops linearly off from the `maxutil` to `maxutil-summitdelta`, and in the second stage it drops off linearly from `maxutil-summitdelta` to `minutil`. The function has the form:

$$f(x) = \begin{cases} f_1(x) & (\texttt{summit} - \texttt{peakwidth}) \leq x \leq \texttt{summit}, \\ f_2(x) & (\texttt{summit} - \texttt{peakwidth} - \texttt{basewidth}) \leq x < (\texttt{summit} - \texttt{peakwidth}), \\ f_3(x) & \texttt{summit} < x \leq (\texttt{summit} + \texttt{peakwidth}), \\ f_4(x) & (\texttt{summit} + \texttt{peakwidth}) < x \leq (\texttt{summit} + \texttt{peakwidth} + \texttt{basewidth}), \\ \texttt{minutil} & \text{otherwise.} \end{cases}$$

where

$$f_1(x) = (\texttt{maxutil} - \texttt{summitdelta}) + (\texttt{summitdelta} * ((x - (\texttt{summit} - \texttt{peakwidth}))/\texttt{peakwidth}))$$
$$f_2(x) = \texttt{minutil} + ((\texttt{maxutil} - \texttt{minutil} - \texttt{summitdelta}) * ((x - (\texttt{summit} - \texttt{peakwidth} - \texttt{basewidth}))/\texttt{basewidth}))$$
$$f_3(x) = (\texttt{maxutil} - \texttt{summitdelta}) + (\texttt{summitdelta} * (((\texttt{summit} + \texttt{peakwidth}) - x)/\texttt{peakwidth}))$$
$$f_4(x) = \texttt{minutil} + ((\texttt{maxutil} - \texttt{minutil} - \texttt{summitdelta}) * (((\texttt{summit} + \texttt{peakwidth} + \texttt{basewidth}) - x)/\texttt{basewidth}))$$

To correlate the above five cases above with the six pieces in Figure 13, $f_1(x)$ is piece `#3`, $f_2(x)$ is piece `#2`, $f_3(x)$ is piece `#4`, $f_4(x)$ is piece `#5`, `minutil` for pieces `#1` and `#6`. The two stage linear drop-off in utility is there to allow the shape of the function to approximate convex or concave functions as shown in Figure 14.

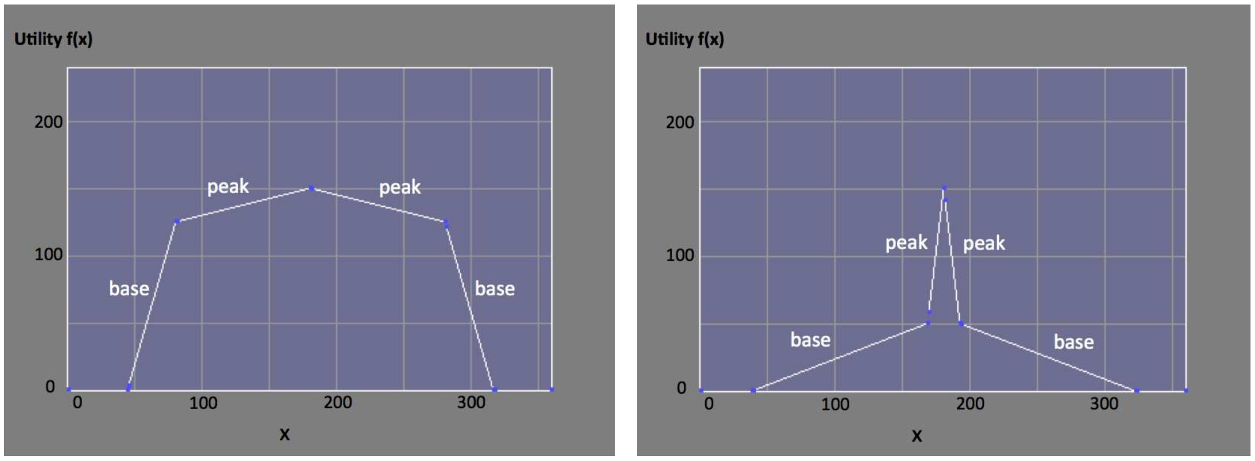


Figure 14: **The `ZAIC_PEAK` tool:** A convex uni-modal function (left) and a non-convex uni-modal function (right).

7.1.3 The `ZAIC_PEAK` Interface Implementation

The following functions define the interface to the `ZAIC_PEAK` tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested.

Many of the below functions take an optional `index` parameter. This is used for creating functions with multiple modes or peaks as described in Section 7.1.6. The default value is zero, or the zeroth index, when only one mode or peak is being implemented. If the given index references a non-existing component, this is considered a fatal configuration error. Example usage is provided in Listing 17 on page 66.

`bool setSummit(double val, int index=0):` Sets the `summit` value of the component at the given index. If no index parameter is provided, the index is zero. If the summit value is outside the range of the domain, it is clipped to the appropriated end. For example if the domain were $[0, 359]$ as the example in Figure 13, and the requested summit value were 550, the summit parameter would be set to 359. This is therefore not regarded as a fatal configuration error, but a warning would be generated anyway. This returns `false` only if index referencing a non-existent component is provided.

`bool setPeakWidth(double val, int index=0):` Sets the peakwidth value of the component at the given index. If no index parameter is provided, the index is zero.

`bool setBaseWidth(double val, int index=0):` Sets the basewidth value of the component at the given index. If no index parameter is provided, the index is zero.

`bool setSummitDelta(double val, int index=0):` Sets the `summitdelta` value of the component at the given index. If no index parameter is provided, the index is zero. A fatal error is declared and `false` is returned if the index is out of range, or if the value is less than zero. Otherwise `true` is returned. If the sumitdelta value is greater than the range determined by `maxutil - minutil`, this is not interpreted as a fatal error, but the `summitdelta` is clipped to the range.

`bool setMinMaxUtil(double min, double, max, int index=0):` Sets the minutil and maxutil values of the component at the given index. If no index parameter is provided, the index is zero. A fatal error is declared and `false` is returned if the index is out of range, or if the min value is greater than or equal to the max value. Otherwise `true` is returned. If the existing `summitdelta` value is greater than the range determined by `maxutil - minutil`, this is not interpreted as a fatal error, but the `summitdelta` is clipped to the range.

`bool setParams(double summit, double peakwidth, double basewidth, double summitdelta, double minutil, double maxutil, int index=0):` Sets the six configuration parameters `summit`, `peakwidth`, `basewidth`, `summitdelta`, `minutil` and `maxutil` all at once. If the `summitdelta` value is greater than the range determined by `maxutil - minutil`, this is not interpreted as a fatal error, but the `summitdelta` is clipped to the range.

`void setSummitInsist(bool val):` Sets the `summitinsist` flag to the given value. The default is `true`. See Section 7.1.4 for more on this parameter.

`void setValueWrap(bool val):` Sets the `valuewrap` flag to the given value. The default is `false`. See Section 7.1.4 for more on this parameter.

`int addComponent():` Allocates a new component and returns the index of the new component. Since the first component exists upon `ZAIC` creation, the first call to this function will return 1, and will result in the `ZAIC_PEAK` instance having two components at index 0 and 1. The default values for the new component are `summit`=0, `peakwidth`=0, bwid=0, sdelta=50, `minutil`=0, `maxutil`=100.

`IvPFunction* extractIvPFunction(bool maxval=true):` This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. If a fatal error was detected in prior parameter setting attempts, this function will simply return the NULL pointer. When the IvP function is extracted from the ZAIC, an `IvPFunction` instance is created from the heap that needs to be later deleted. The ZAIC tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.

`string getWarnings():` When or if fatal (or non-fatal) problems are encountered in setting the parameters, the tool appends a message to a local warning string. This string can be retrieved by this function. A non-empty string does not necessarily mean a fatal configuration error was encountered. Instead, the `stateOK()` function below should be consulted.

`bool stateOK():` This function returns `true` if no fatal errors were encountered during configuration attempts, otherwise it returns `false`. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the `getWarnings()` function above can be consulted.

7.1.4 The Value-Wrap and Summit-Insist Parameters

Two additional Boolean parameters may be set for an instance of `ZAIC_PEAK`. They are the `valuewrap` and `summitinsist` parameters set with the following functions with the defaults shown:

```
zaic.setValueWrap(false);     // Default value is false
zaic.setSummitInsist(true);   // Default value is true
```

When the `valuewrap` parameter is `true`, the utility associated with the domain variable value "wraps" around. For example, if the domain variable is the vehicle *heading*, which may have the domain $[0, 359]$, a value of 10 degrees is evaluated as being only 20 degrees different from 350, rather than being different by 330 degrees. The two functions depicted in Figure 15 differ only in the setting of the `valuewrap` parameter.

The value of the `summitinsist` parameter can affect the generated objective function in the following two scenarios. In the first case, consider the domain to be the possible headings with 360 discrete choices $[0, 359]$, and the `peakwidth` and `basewidth` are both zero. If the `summit` were then set to 90.25, one way to interpret this is that all 360 discrete heading choices have a utility of zero, since none are equal exactly to 90.25. This is the interpretation when `summitinsist` is

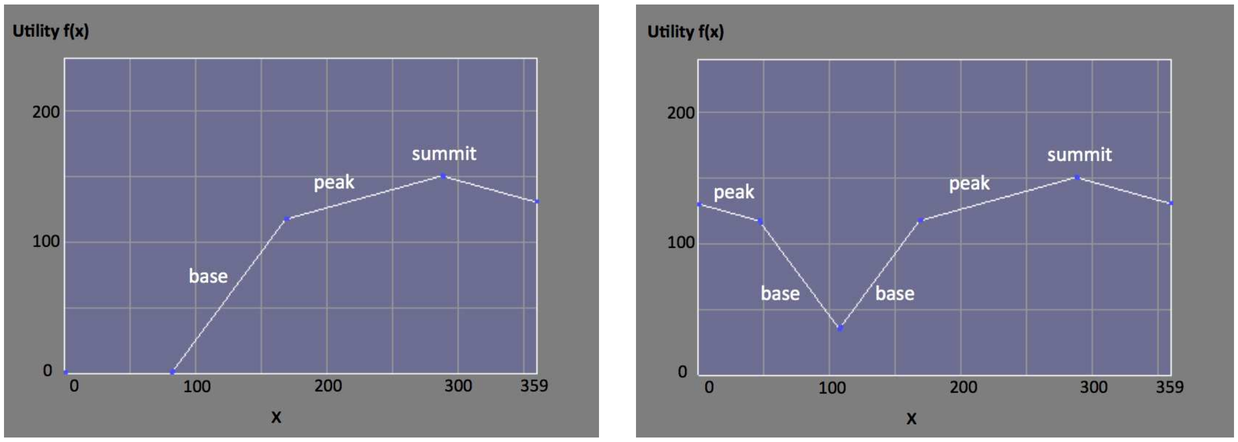


Figure 15: **The `valuewrap` parameter in the `ZAIC_PEAK` tool:** A function generated with `valuewrap=false` on the left, and function generated with `valuewrap=true` and otherwise identical parameters on the right.

false. When set to `true`, the same set of parameters would generate an objective function that ranked the heading of 90 degrees with maximum utility and all other heading choices with the minimum utility (an IvP function with 3 pieces, i.e., intervals). In the second case, consider the domain to be possible speeds with 31 discrete choices $[0, 3.0]$, with the `summit` set to 4.0 with a peakwidth and basewidth of 0.25. The `ZAIC_PEAK` tool would generate an objective function ranking all speeds equally with the minimum utility when `summitinsist` is set to `false`. When set to `true`, the `ZAIC_PEAK` tool would generate an objective function ranking the highest speed in the domain (3.0) with maximum utility (`maxutil`), and all other speeds with minimum utility (`minutil`). The default setting is `summitinsist=true` since this seems the more reasonable thing to do in most such cases.

7.1.5 Using the `ZAIC_PEAK` Tool

Usage of the `ZAIC_PEAK` tool boils down to the following four steps.

- Step 1: Create the IvP domain, or retrieve it if otherwise already created.
- Step 2: Create the `ZAIC_Peak` instance with a domain and domain variable.
- Step 3: Set the `ZAIC_Peak` parameters.
- Step 4: Extract the IvP function.

A code example of the four steps is provided in Listing 17 below. This code example describes a function that builds and returns an IvP function using the `ZAIC_LEQ` tool. It is not too different from the activity inside a typical implementation of `onRunState` in an IvP behavior.

Listing 17 - Example usage of the `ZAIC_Peak tool` *corresponding to Figure 13.*

```
IvPFunction *buildIvPFunction()
{
  // Step 1 - Create the IvPDomain, the function's domain
```

```
 3    IvPDomain domain;
 4    domain.addVar("depth", 0, 600, 601);
 5
 6    // Step 2 - Create the ZAIC_PEAK with the domain and variable name
 7    ZAIC_PEAK  zaic_peak(domain, "depth");
 8
 9    // Step 3 - Configure the ZAIC_LEQ parameters
10    zaic_peak.setSummit(150);
11    zaic_peak.setMinMaxUtil(20, 120);
12    zaic_peak.setBaseWidth(60);
13
14    // Step 4 - Extract the IvP function
15    IvPFunction *ivp_function = 0;
16    if(zaic_leq.stateOK())
17      ivp_function = zaic_peak.extractIvPFunction();
18    else
19      cout << zaic_peak.getWarnings();
20    return(ivp_function)
```

The lines comprising step 4 (lines 15-20) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

```
return(zaic_peak.extractIvPFunction());
```

This is simpler, but the warning information is potentially useful. When the `ZAIC_PEAK` tool is used within an IvP behavior, the warnings can be posted to the MOOSDB in the variable `BHV_WARNING` which is monitored by other tools.

7.1.6 Support for Multi-Modal Functions with the `ZAIC_PEAK` Tool

The `ZAIC_PEAK` tool will allow additional components to be added to provide a multi-modal effect. A *component* refers to the set of six parameters `summit`, `peakwidth`, `basewidth`, `summitdelta`, `minutil`, `maxutil`. Adding a second component creates a multi-modal function as shown in Figure 16.

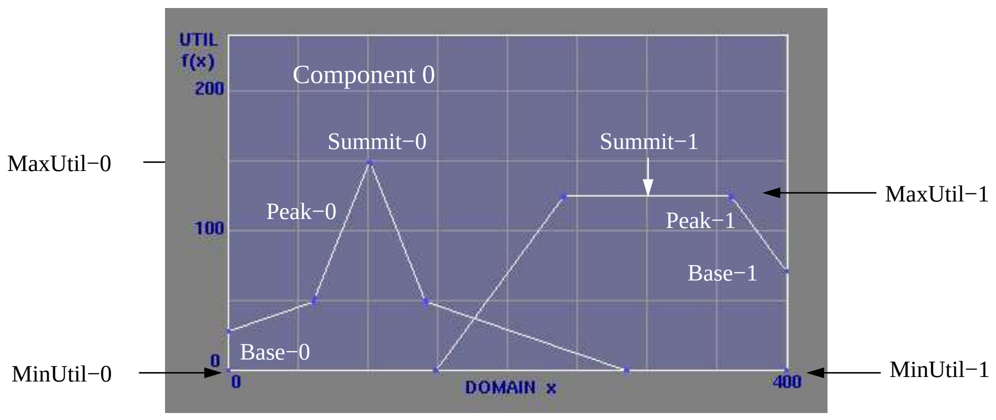


Figure 16: **Multiple modes with the ZAIC_PEAK tool**: additional components can be added to create a multi-modal objective function. Each component is comprised of the six parameters `summit`, `peakwidth`, `basewidth`, `summitdelta`, `minutil`, and `maxutil`.

Listing 18 shows how this is done. In lines 2-3, the `ZAIC_PEAK` instance is created and the parameters are set for the first component. A second component is allocated in line 5 with the call to `addComponent()` which returns the index of the newly created component. This index is passed as the last argument to the function call on line 6 to clarify that the parameters are to be applied to the second component (at index 1).

Listing 18 - Using the `ZAIC_PEAK` *tool to build and return a multi-mode IvP Function.*

```
 0  IvPFunction *buildIvPFunction(IvPDomain domain, string varname)
 1  {
 2    ZAIC_PEAK zaic(domain, varname);
 3    zaic.setParams(300, 80, 100, 15, 0, 100);  // No index given - assumed to be zero.
 4
 5    int index = zaic.addComponent();
 6    zaic.setParams(600, 130, 35, 30, 0, 147, index); // Last parameter is component index
 7
 8    zaic.setValueWrap(false);     // Not component specific - no index given
 9    zaic.setSummitInsist(true);   // Not component specific - no index given
10    bool take_the_max = true
11    IvPFunction *ipf = zaic.extractIvPFunction(take_the_max);
12    return(ipf);
13  }
```

When the IvP function is extracted from the `ZAIC_PEAK`, in line 9, the composition of the multiple components can be interpreted in one of two ways - by taking the *maximum* or the *sum* of the two components. In the former, the combined utility is the maximum of the values given by the individual components. In the latter the combined utility is the sum of the individual components. The `extractIvPFunction(bool)` function on line 11 will return a composition based on taking the max if passed `true`, and will take the sum otherwise. The default (if no value is passed) is `true`. The difference between the two extractions is shown in Figure 17 for the two components shown in Figure 16.

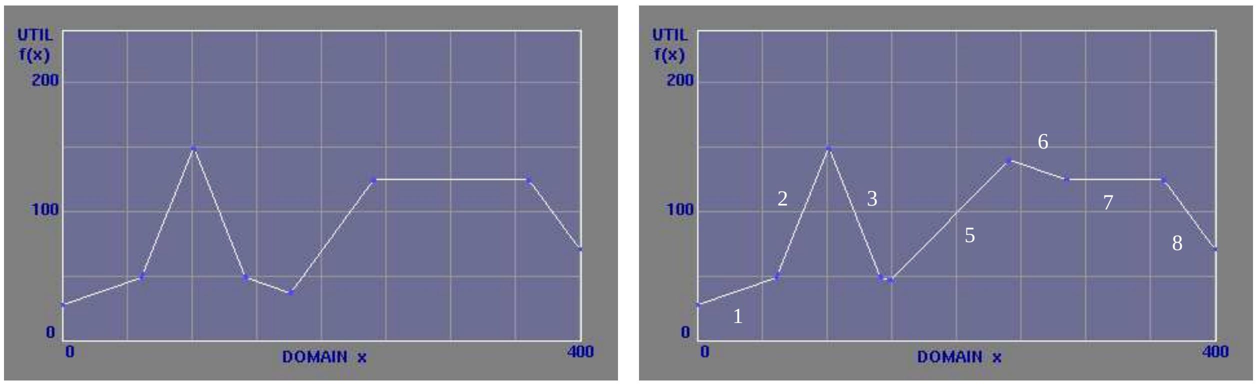


Figure 17: **Options for combining components**: On the left is the result of combining the two components in Figure 16 by using the max value from the two components. On the right is the result of combining the two components by using the sum of the values from the two components.

All component parameter-setting functions defined for the `ZAIC_PEAK` class take an optional final argument indicating the intended component index. If the argument is not provided, it is assumed to be zero, the index of the one component created automatically when the `ZAIC_PEAK` instance is created. The default values of a newly added component are are `summit`=0, `peakwidth`=0, `basewidth`=0, `summitdelta`=50, `minutil`=0, `maxutil`=100. The `valuewrap` and `summitinsist` parameters are not component-specific parameters, and therefore are only called once on a particular `ZAIC_PEAK` instance, and do not have an optional index parameter.

7.2 The `ZAIC_LEQ` and `ZAIC_HEQ` Tools

7.2.1 Brief Overview

The `ZAIC_LEQ` tool is used for generating IvP functions where there is a constant, maximum utility associated with a decision variable whose value is kept *less than or equal* to (LEQ) a given value. For example if a UUV component is known to work reliably up to a certain depth, or if a vehicle's speed is to be kept below a certain value to prevent interference with another sensor or communications equipment. The `ZAIC_LEQ` tool allows for expressing a linear drop-off in utility between two values. For example, if the fictional UUV component is rated to a depth of 200 meters, the utility function may have a maximum utility for depths up to 150 meters and minimum utility at 210 meters as in the example in Figure 18.

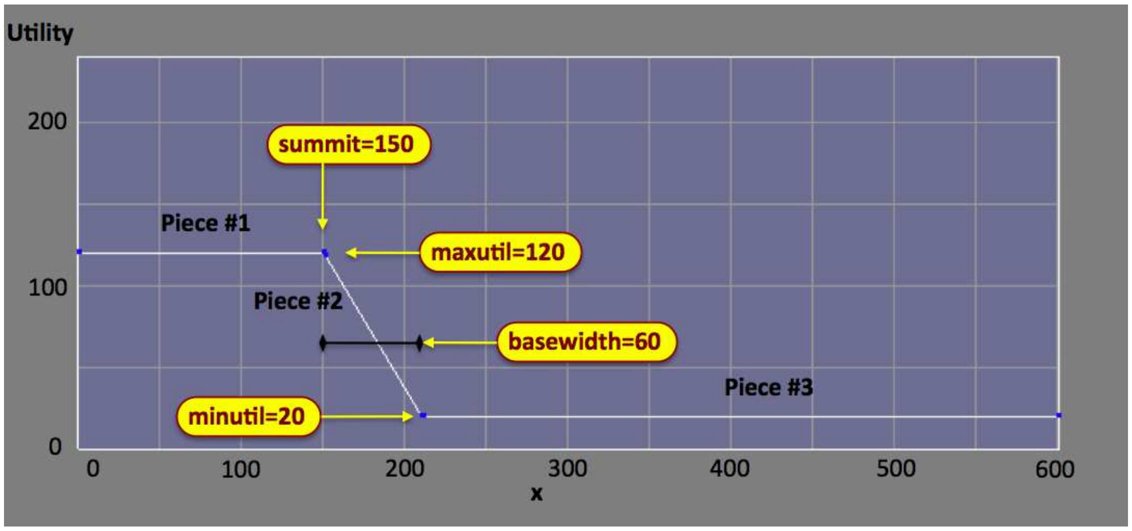


Figure 18: **The `ZAIC_LEQ` tool**: facilitates building simple 2-3 piece piecewise defined utility functions over a single decision variable whose value is to be kept less than or equal to a given value. It accepts four parameters, the `summit`, `minutil`, `maxutil`, `basewidth`. In the figure `summit`= 150, `minutil`= 20, `maxutil`= 120, `basewidth`= 60.

Likewise, the `ZAIC_HEQ` tool is used for generating objective functions where there is a constant, maximum utility associated with a decision variable whose value is kept *greater than or equal* to (HEQ) a given value. These two tools have a similar interface. Most of what is described below for one tool applies to the other. The differences are distinguished in Section 7.2.5. The IvP functions generated by these ZAIC tools have a small footprint, having either two or three pieces.

7.2.2 The ZAIC_LEQ Parameters and Function Form

The ZAIC_LEQ tool accepts four parameters in defining $f(x)$. The summit parameter is the point where maximum utility begins to drop off. The minutil parameter is the minimum value of $f(x)$, with a default value of zero. The maxutil is the maximum value of $f(x)$, with a default value of 100. The function has the form:

$$f(x) = \begin{cases} \texttt{maxutil} & x \leq \texttt{summit}, \\ \texttt{minutil} & \text{otherwise.} \end{cases}$$

The basewidth parameter can be used to soften the drop in utility as shown in Figure 18. When basewidth has the default value of zero, the general form is as above. When basewidth is configured with a positive value, the general form is:

$$f(x) = \begin{cases} \texttt{maxutil} & x \leq \texttt{summit}, \\ \texttt{minutil} + ((\texttt{maxutil - minutil}) \text{ * } ((\text{x - }\texttt{summit}) \text{ / } \texttt{basewidth})) & \texttt{summit} < x \leq \texttt{summit} + \texttt{basewidth}, \\ \texttt{minutil} & \text{otherwise.} \end{cases}$$

For the example in Figure 18, the function is given below.

$$f(x) = \begin{cases} 120 & x \leq 150, \\ 20 + ((120 - 20) * ((210 - x)/(210 - 150))) & 150 < x \leq 210, \\ 20 & \text{otherwise.} \end{cases}$$

The three above cases corresponds to the three pieces generated for the IvP function shown in Figure 18.

7.2.3 The ZAIC_LEQ Interface Implementation

The following functions define the interface to the ZAIC_LEQ tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested. Example usage is provided in Listing 19.

ZAIC_LEQ(IvPDomain domain, string varname): The constructor takes two arguments, an IvP domain and a variable name contained in the domain. The named variable needs to be just *one* of the variables used in the IvP domain; not necessarily the only one. If the named variable is not part of the IvP domain, this is regarded as a fatal error.

bool setSummit(double summit): Sets the summit value. If the summit value is outside the range of the domain, it is clipped to the appropriated end. For example if the domain were $[0, 600]$ as the example in Figure 18, and the requested summit value were 650, the summit parameter would be set to 600. This is therefore not regarded as a fatal configuration error, but a warning would be generated anyway. This function always returns `true`.

`bool setMinMaxUtil(double min, double max)`: Sets the values for `minutil` and `maxutil`. If the `minutil` is greater or equal to `maxutil`, this is regarded as a fatal configuration error, a warning is generated, and the function returns `false`.

`bool setBaseWidth(double basewidth)`: Sets the `basewidth` parameter value. The given value must be greater than or equal to zero. Otherwise this is regarded as a fatal configuration error, a warning is generated, and the function returns `false`.

`IvPFunction *extractIvPFunction()`: This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. If a fatal error was detected in prior parameter setting attempts, this function will simply return the NULL pointer. When the IvP function is extracted from the ZAIC, an `IvPFunction` instance is created from the heap that needs to be later deleted. The ZAIC tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.

`string getWarnings()`: When or if fatal (or non-fatal) problems are encountered in setting the parameters, the tool appends a message to a local warning string. This string can be retrieved by this function. A non-empty string does not necessarily mean a fatal configuration error was encountered. Instead, the `stateOK()` function below should be consulted.

`bool stateOK()`: This function returns `true` if no fatal errors were encountered during configuration attempts, otherwise it returns `false`. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the `getWarnings()` function above can be consulted.

7.2.4 Using the `ZAIC_LEQ` Tool

Usage of the `ZAIC_LEQ` tool boils down to the following four steps.

- Step 1: Create the IvP domain, or retrieve it if otherwise already created.
- Step 2: Create the `ZAIC_LEQ` instance with a domain and domain variable.
- Step 3: Set the `ZAIC_LEQ` parameters.
- Step 4: Extract the IvP function.

A code example of the four steps is provided in Listing 19 below. This code example describes a function that builds and returns an IvP function using the `ZAIC_LEQ` tool. It is not too different from the activity inside a typical implementation of `onRunState` in an IvP behavior.

Listing 19 - Example usage of the `ZAIC_LEQ tool` corresponding to Figure 18.

```
 0  IvPFunction *buildIvPFunction()
 1  {
 2    // Step 1 - Create the IvPDomain, the function's domain
 3    IvPDomain domain;
 4    domain.addVar("depth", 0, 600, 601);
 5
 6    // Step 2 - Create the ZAIC_LEQ with the domain and variable name
 7    ZAIC_LEQ  zaic_leq(domain, "depth");
 8
 9    // Step 3 - Configure the ZAIC_LEQ parameters
10    zaic_leq.setSummit(150);
11    zaic_leq.setMinMaxUtil(20, 120);
12    zaic_leq.setBaseWidth(60);
13
14    // Step 4 - Extract the IvP function
15    IvPFunction *ivp_function = 0;
16    if(zaic_leq.stateOK())
17      ivp_function = zaic_leq.extractIvPFunction();
18    else
19      cout << zaic_leq.getWarnings();
20    return(ivp_function)
```

The lines comprising step 4 (lines 15-20) are conservative in that they first check to see if no fatal configuration errors were encountered, and writes the warnings to the terminal if found. It does not even attempt to extract an IvP function if an error was encountered. These five lines could have been replaced by one line:

```
return(zaic_leq.extractIvPFunction());
```

This is simpler, but the warning information is potentially useful. When the `ZAIC_LEQ` tool is used within an IvP behavior, the warnings can be posted to the MOOSDB in the variable `BHV_WARNING` which is monitored by other tools.

7.2.5 The `ZAIC_HEQ` Tool

Like the `ZAIC_LEQ` tool, the `ZAIC_HEQ` too is used for generating objective functions where there is a constant, maximum utility associated with a decision variable whose value is kept *greater than or equal* to a given value. The parameters described in Section 7.2.2 and interface implementation described in Section 7.2.3 for the `ZAIC_LEQ` tool are identical for the `ZAIC_HEQ` tool. The parameters are interpreted differently however. The same parameters used in Figure 18 are used in the `ZAIC_HEQ` tool to give the function shown in Figure 19.

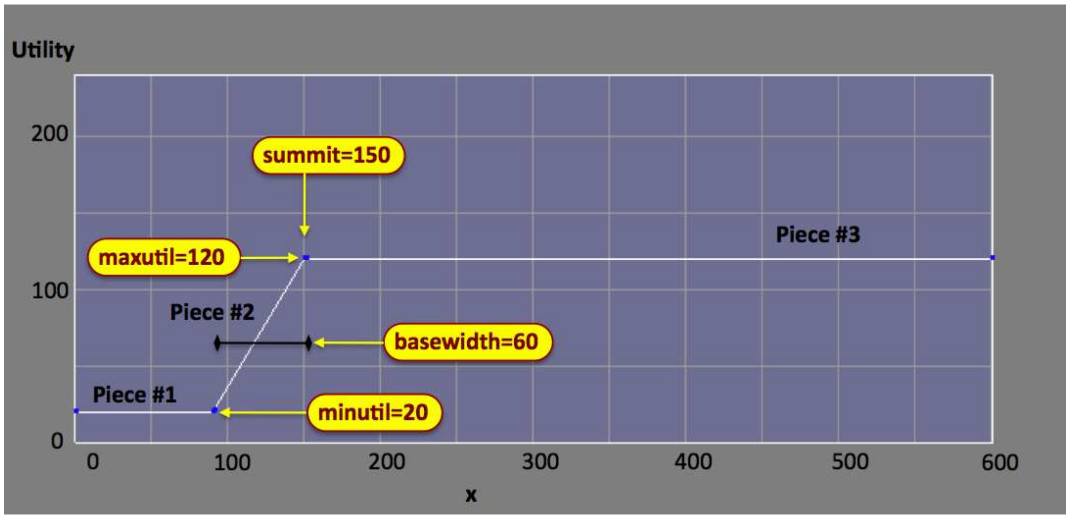


Figure 19: **The `ZAIC_HEQ` tool**: facilitates building simple 2-3 piece piecewise defined utility functions over a single decision variable whose value is to be kept greater than or equal to a given value. It accepts four parameters, the `summit`, `minutil`, `maxutil`, `basewidth`. In the figure `summit`= 150, `minutil`= 20, `maxutil`= 120, `basewidth`= 60.

7.2.6 A Warning about the Maximum Utility Plateau

It is worth noting a potential pitfall regarding the maximum utility plateau generated by both the `ZAIC_LEQ` and `ZAIC_HEQ` tools. By having equal utility for all domain values in the plateau range, no one domain value is preferred. If this is the only IvP objective function involved in the decision process *for the particular domain variable*, it is not clear what value will be chosen by the IvP solver. Even more troublesome is that the chosen value may change between iterations giving the appearance that the decision engine is thrashing. In this case the solver is simply faithfully and strictly interpreting the problem it was given. In short, these objective functions are designed to work in conjunction with others that express preferences in a non-plateau manner.

8 The Reflector Tool for Building N-Variable IvP Functions

8.1 Overview

The IvPBuild Toolbox contains the `Reflector` tool for building IvP functions over $n \geq 2$ decision variables. Although the tools work with $n = 1$ variables, the `ZAIC` tools are typically used instead. The `Reflector` tool operates on a particular division of labor. The user of the `Reflector` provides a black-box function implementation able to provide a utility value for any queried input. The possible queries are limited to the domain or decision space of the function expressed with an `IvPDomain` instance. This black-box routine in essence is the underlying objective function to be approximated by the generated IvP function (Figure 20).

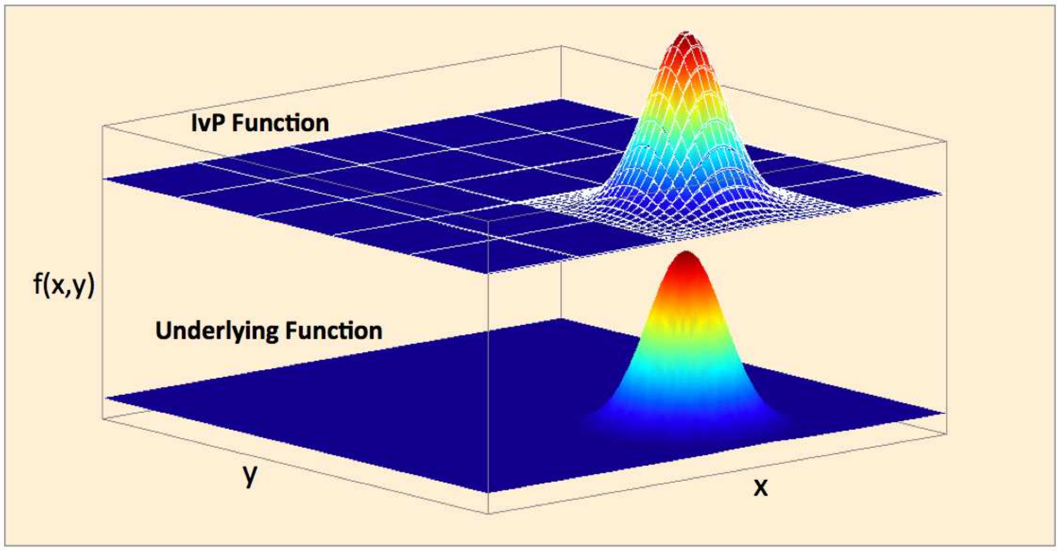


Figure 20: **The Reflector Tool**: An IvP function approximates an underlying function $f(x, y)$ using a piecewise linear structure with 698 pieces. The piece distribution need not be uniform allowing greater resolution over parts of the domain where the function is detected to be less locally linear.

The goal is to generate an acceptable IvP function approximation by querying the underlying function for as a small subset of the total function domain as possible. The CPU time taken to evaluate the underlying function can easily be the most expensive part of building the IvP function for a given behavior. Implementing the underlying function efficiently and in a way that accurately reflects the intent of the behavior can be the most challenging part of building a behavior. The `Reflector` tool can be used with a very simple interface that builds an IvP function given a pointer to the underlying function and the number of pieces to use in the IvP function. Such IvP functions will be constructed with pieces of uniform size. This is discussed in Section 8.3. The `Reflector` can be configured with more advanced parameters to build an IvP function with non-uniformly distributed pieces as depicted in Figure 20. These methods are used to build functions that more accurately approximate their underlying functions *and* use less pieces. The advanced `Reflector` parameters are discussed in Section 9.

8.2 Implementing Underlying Functions within the `AOF` Class

The primary job of a behavior author is to provide a method capable of evaluating any candidate decision in the decision space. Evaluating each decision can be prohibitively time consuming and a piecewise linear approximation with an IvP function is typically built by invoking the evaluation function for only a small subset of the domain. The build toolbox depends on access to the evaluation routine in a generic way, as a pointer to an instance of the class `AOF`, the "actual objective function".

8.2.1 The `AOF` Class Definition

The `AOF` class itself is abstract, and the AOF pointer actually points to an implemented subclass with a few key virtual functions overloaded. The `AOF` class definition (slightly simplified) is given in Listing 20.

Listing 20 - The `AOF` *class definition.*

```
#include "IvPBox.h"
#include "IvPDomain.h"
class AOF{
public:
  AOF(IvPDomain domain) {m_domain=domain;};
  virtual ~AOF() {};

  virtual double evalPoint(vector<double>);
  virtual bool   setParam(string, double)  {return(false);};
  virtual bool   setParam(string, string)  {return(false);};
  virtual bool   initialize()              {return(true);};

  double extract(string, const vector<double>&) const;

protected:
  IvPDomain m_domain;
};
#endif
```

This is essentially a template for a function, defined over the domain given in the constructor. The mapping from the domain to a range is implemented in the `evalPoint()` function which takes a vector of numerical values representing a candidate decision in the IvP domain decision space. The `setParam()` and `initialize()` virtual functions provide a generic way for subclasses to set their parameters.

8.2.2 An Example Underlying Function Implemented as an `AOF` Subclass

As an example consider the simple linear function $f(x, y) = m \cdot x + n \cdot y + b$, implemented by the class `AOF_Linear` shown in Listing 21 and 22 below. The class contains three member variables, lines 11-13 in Listing 21 for representing the coefficient and scalar parameters.

Listing 21 - `AOF_Linear.h` - *The class definition for the* `AOF_Linear` *class.*

```
 0  #include "AOF.h"
 1  class AOF_Linear: public AOF {
 2  public:
 3    AOF_Linear(IvPDomain domain) : AOF(domain)
 4     {m_coeff = 0; n_coeff=0; b_scalar=0;};
 5    ~AOF_Linear() {};
 6
 7    double evalPoint(vector<double>);
 8    bool   setParam(const string& param, double val);
 9
10  private:
11    double m_coeff;
12    double n_coeff;
13    double b_scalar;
14  };
```

The class implementation is shown in Listing 22. The constructor takes an instance of `IvPDomain` as an argument and passes it to the `AOF` superclass for handling (line 1). The m and n coefficients are set in the `setParam()` function and will return `true` if either the `mcoeff`, `ncoeff`, or `bscalar` parameters are passed, and `false` otherwise.

Listing 22 - `AOF_Linear.cpp` - *The class implementation for the* `AOF_Linear` *class.*

```
 0  //-------------------------------------------------------------
 1  bool AOF_Linear::setParam(string param, double val)
 2  {
 3    if(param == "mcoeff")
 4      m_coeff = val;
 5    else if(param == "ncoeff")
 6      n_coeff = val;
 7    else if(param == "bscalar")
 8      b_scalar = val;
 9    else
10      return(false);
11    return(true);
12  };
13
14  //-------------------------------------------------------------
15  double AOF_Linear::evalPoint(vector<double> point)
16  {
17    double x_val = extract(‘‘x’’, point);
18    double y_val = extract(‘‘y’’, point);
19
20    return((m_coeff * x_val) + (n_coeff * y_val) + b_scalar);
21  }
```

The `evalPoint()` function in lines 15-21 is where the actual implementation of $f(x, y) = m \cdot x + n \cdot y + b$ is implemented (on line 20). The argument to this function is a vector of values holding the values for the x and y variables. The ordering of these values i.e., which of the two variables, x or y, is contained in the first value of the vector, is sorted out in the two calls to the `extract()` function in lines 17-18. This sorting out is possible because the ordering is determined by the `IvPDomain` member variable defined at the `AOF` superclass level, and provided in the constructor. The `AOF_Linear` class, used as an example here, is included in the code distribution in `lib_ivpbuild`. It may serve as a template in building a new `AOF_YourAOF` class. It is also can be used to verify that the build tools will work in the extreme case of creating a piecewise function with only one piece. And in the case of `AOF_Linear` the piecewise "approximation" is exact.

8.2.3 Another `AOF` Example Class Implementation for Gaussian Functions

A second function type, implementing Gaussian functions, is implemented as the `AOF_Gaussian` class in the IvP Toolbox in the `lib_ivpbuild` module. This function is a bit more interesting in that a piecewise linear approximation needs multiple pieces to generate a fairly good approximation (understanding that "fairly good" is subjective). It is also interesting in that, depending on the configuration, there may be large portions of the function that are indeed locally linear and in need of relatively few pieces to generate a decent approximation. This function will be used extensively in later examples of usage and performance of the `Reflector` tool. The Gaussian function form is given by:

$$f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})} \tag{1}$$

The function is defined over the two variables, x and y, and has four parameters. Examples for two groups of parameter settings are shown in Figure 21 on page 78, and in Figure 22 on page 80. The coefficient A is the amplitude, x_0 and y_0 are the center, and σ represents the spread of the blob. This function is implemented by the class `AOF_Gaussian` shown in Listings 23 and 24. Note that it is a subclass of the `AOF` class and overrides the critical function `evalPoint()`. It also implements the `setParam()` function for setting the four parameters in (1).

Listing 23 `AOF_Gaussian.h` - *The class definition for the* `AOF_Gaussian` *class.*

```
class AOF_Gaussian: public AOF {
public:
  AOF_Gaussian(IvPDomain domain) : AOF(domain)
    {m_xcent=0; m_ycent=0; m_sigma=1; m_range=100;};
  ~AOF_Gaussian() {};

  double evalPoint(vector<double> point);
  bool   setParam(string param, double value);

private:
  double  m_xcent;
  double  m_ycent;
  double  m_sigma;
  double  m_range;
};
```

Listing 24 `AOF_Gaussian.cpp` - *The class implementation for the* `AOF_Gaussian` *class.*

```
//----------------------------------------------------------------
// Procedure: setParam

bool AOF_Gaussian::setParam(string param, double value)
{
  if(param == "xcent")      m_xcent = value;
  else if(param == "ycent") m_ycent = value;
  else if(param == "sigma") m_sigma = value;
  else if(param == "range") m_range = value;
  else
    return(false);
  return(true);
}
```

```

//----------------------------------------------------------------
// Procedure: evalPoint

double AOF_Gaussian::evalPoint(vector<double> point)
{
  double xval = extract("x", point);
  double yval = extract("y", point);
  double dist = hypot((xval - m_x9ent), (yval - m_ycent));
  double pct  = pow(M_E, -((dist*0ist)/(2*(m_sigma * m_sigma))));

  return(pct * m_range);
}
```

An example is shown in Figure 21 below. The domain for both the x and y variables is $[-250, 250]$ containing 501 x 501 = 251,001 points.

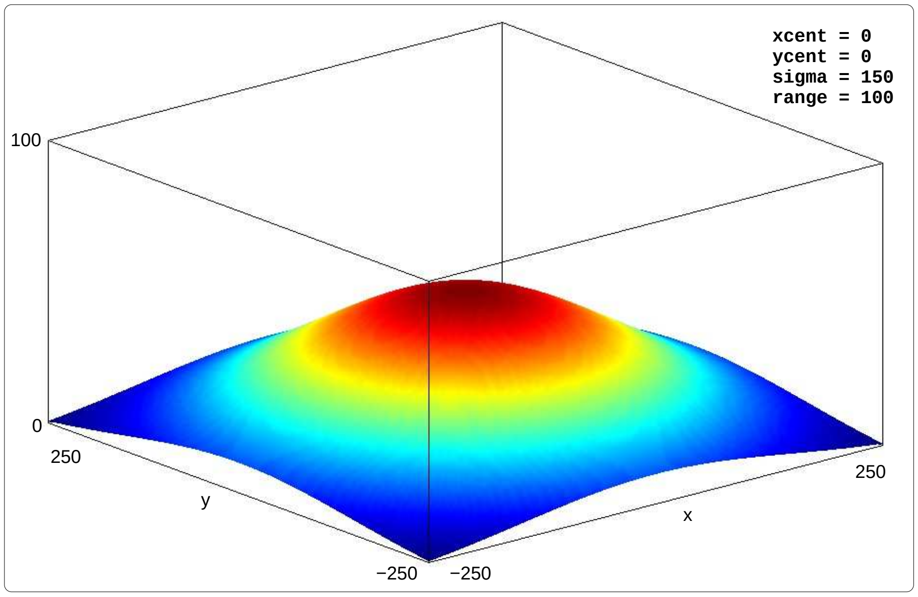


Figure 21: **A Gaussian function**: A rendering of the function $f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where A = `range` = 100, σ = `sigma` = 150, x_0 = `xcent` = 0, y_0 = `ycent` = 0. The domain for x and y ranges from -250 to 250.

8.3 Basic Reflector Tool Usage Tool with Examples

Using the `Reflector` tool boils down to the four steps below. The third step may be non-existent if the user is building simple uniform functions.

- Step 1: Create the underlying function, `AOF` instance, and set its parameters.
- Step 2: Create the `Reflector` instance passing it a pointer to the AOF instance.
- Step 3: Set parameters for the `Reflector` if necessary or desired.
- Step 4: Direct the `Reflector` to build the IvP function and then extract it.

A code example of the four steps is provided in Listing 25 below. This code example describes a function that builds and returns an IvP function using the `Reflector` tool. It is not too different from the activity inside a typical implementation of `onRunState` in an IvP behavior.

Listing 25 - An example use of the Reflector to create a uniform IvP function.

```
 0  IvPFunction *buildIvPFunction(IvPDomain ivp_domain)
 1  {
 2    // Step 1 - Create the AOF instance and set parameters
 3    AOF_Gaussian aof(ivp_domain);
 4    aof.setParam("xcent", 50);
 5    aof.setParam("ycent", -150);
 6    aof.setParam("sigma", 32.4);
 7    aof.setParam("range", 150);
 8
 9    // Step 2 - Create the Reflector instance given the AOF
10    OF_Reflector reflector(&aof);
11
12    // Step 3 - Parameterize the Reflector (None in this case)
13
14    // Step 4 - Build and Extract the IvP Function
15    int  amt_created = reflector.create(1000);
16    IvPFunction *ipf = reflector.extractIvPFunction();
17
18    cout << ‘‘Pieces in the new IvPFunction: ‘‘ << amt_created << endl;
19    return(ipf);
20  }
```

The underlying function is created on lines 3-7 creating the Gaussian function with parameters shown in Figure 22. The `Reflector` is created on line 10 with a pointer to the new Gaussian underlying function. In lines 15-16, the `Reflector` creates and returns the IvP function. In this simple style of usage, no parameters are set on the `Reflector` after it is created. The result will be an IvP function with uniform piece shape, where the total number of pieces are requested on line 15. (Note that 1000 pieces are requested, but not all requested piece counts are feasible or practical. See Section 9.2.2 for more on this). The requested number of uniform pieces affects three practical metrics of the resulting the IvP function. The error in its representation of the underlying function, the time to create the IvP function, and the number of pieces in the IvP function. The goal is to minimize each, but they are in competition with each other.

Figure 23 depicts four IvP function approximations of the same underlying function, and Table 2 illustrates the relationship between the three metrics of (a) piece count, (b) create time, and (c) accuracy in representing the underlying function. The user determines the most appropriate compromise between these metrics for the application at hand. In general, a gain on one metric is traded off against a sacrifice on other metrics. With the additional tools described in Section 9, it is often possible to make improvements in all three metrics simultaneously. One way to look at this is that there is a fourth metric, *ease-of-use*, that can instead be dialed back to achieve gains in all of the first three metrics. In Listing 25, the absence of Step 3, where insightful parameters could have been provided to the `Reflector` to produce non-uniform functions, could be viewed as optimizing the ease-of-use metric.

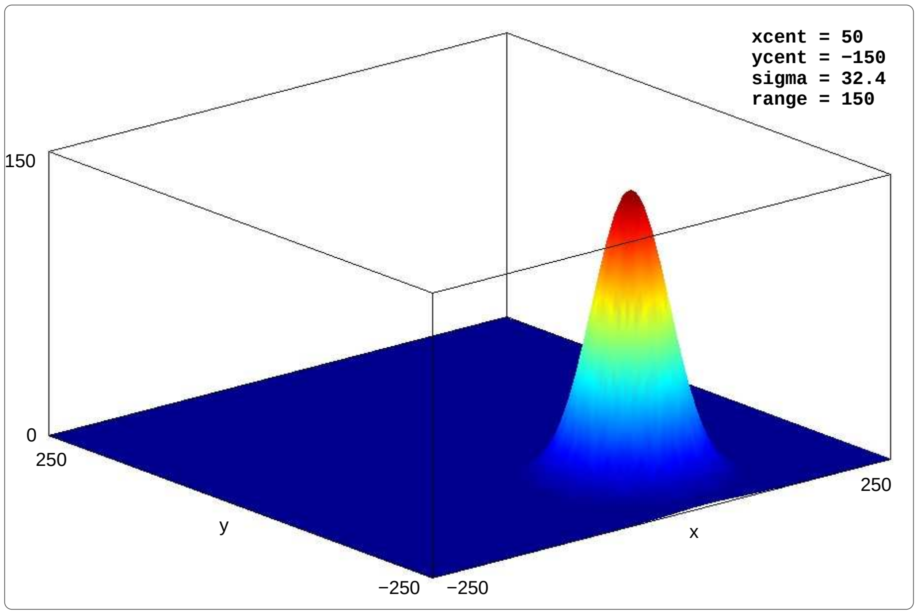


Figure 22: **A Gaussian function**: A rendering of the function $f(x, y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}$ where A = `range` = 150, σ = `sigma` = 32.4, x_0 = `xcent` = 50, y_0 = `ycent` = −150. The domain for x and y ranges from −250 to 250.

Case	Edge	Pieces	Layout	Worst	Avg	Time
	Size			Error	Error	msecs
1	3	27889	(167x167)	0.0761	0.0014	656.4
2	5	1000	(100x100)	0.3019	0.0048	160.0
3	7	5184	(72x72)	0.6720	0.0104	83.3
4	10	2500	(50x50)	1.4589	0.0232	39.9
5	15	1156	(34x34)	3.4532	0.0551	18.9
6	20	625	(25x25)	5.5855	0.1014	10.4
7	25	400	(20x20)	7.79764	0.1585	6.5
8	30	289	(17x17)	12.0347	0.2303	4.7
9	40	169	(13x13)	24.2977	0.3919	2.8
10	50	100	(10x10)	18.2113	0.5917	1.6
11	75	49	(7x7)	42.0652	1.2143	0.9
12	100	25	(5x5)	30.3938	2.0285	0.5

Table 2: **IvP function configurations and metrics**: The relationship between piece size, accuracy and construction time is shown for varying uniform piece size. Four of the row entries are rendered in Figure 23.

8.4 The Full `Reflector` Interface Implementation

The following functions define the interface to the `Reflector` tool. In constructing and setting parameters, the instance maintains a Boolean flag indicating if any fatal configuration errors were detected. In such cases, a warning string is generated for optional retrieval, and the error renders the instance effectively useless, never yielding an IvP function when requested. Example usage is provided in Listing 25 on page 79.

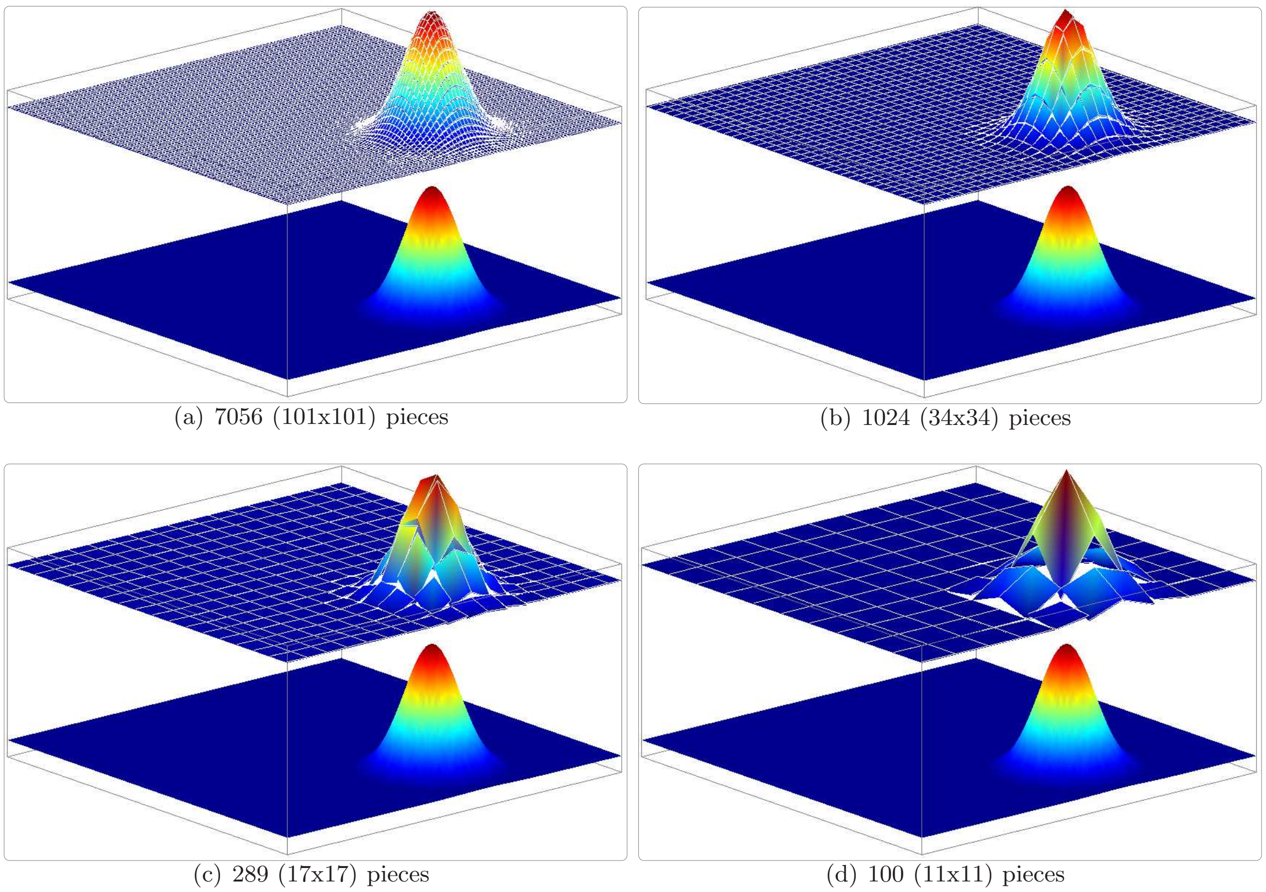


Figure 23: **Four IvP functions approximating the same underlying function**: Each IvP function uses a different number of uniform pieces.

`OF_Reflector(AOF*)`: The constructor takes a single argument, a pointer to the underlying function to be approximated by the Reflector. The `AOF` instance contains an instance of the IvPDomain which will also be the IvPDomain of any IvP functions created with the Reflector.

`int create(int pieces=-1)`: This function generates a new IvP function based on the prevailing parameter settings at the time of invocation. Many of the parameters affecting the form of the function are settable separately in the `setParam()` function, including the parameter specifying the number of pieces. If the optional `pieces` argument is provided in this function call, and if the value of the argument is ≥ 1, this overrides any piece count request set otherwise. This function will create an IvP function that the user can then obtain via the function `extractIvPFunction()` described below. The integer value returned is the number of pieces in the newly created IvP function. A value of zero indicates something has gone wrong.

`IvPFunction *extractIvPFunction()`: This function returns a new IvP function built during a prior invocation of the `create()` function described above. If an error was encountered in either the parameter setting attempts, or in the invocation of the `create()` function, this function will simply return the NULL pointer. When the IvP function is extracted from the Reflector, an

`IvPFunction` instance is created from the heap that needs to be later deleted. The Reflector tool does not delete this. It is the responsibility of the caller. Typically this tool is used within a behavior, and the behavior passes the IvP function to the helm and the helm deletes all IvP functions.

`string getWarnings():` When or if problems are encountered in setting the parameters, the Reflector appends a message to a local warning string. This string can be retrieved by this function.

`bool stateOK():` This function returns `true` if no errors were encountered during configuration attempts, otherwise it returns `false`. If an error has been encountered, this state cannot be reversed. The instance has been rendered effectively useless. To gain insight into the nature of the error, the `getWarnings()` function above can be consulted.

`bool setParam(string param, string value):` This function is used for setting parameters on many optional tools more advanced than specifying the number of pieces to be used in a simple uniform function. An overview is provided here, with more detailed deferred to later sections that cover the advanced tools.

- `uniform_amount`: The amount of pieces to use in the creation of a simple uniform function. Alternatively can be supplied in the call to `create()` as described above.
- `uniform_piece`: A string description of the size and shape of a piece used during the creation of a pure uniform function. Details described in Section 9.1.2.
- `strict_range`: When set to `true`, the range of the linear interior function is guaranteed to stay within the range of any sampled points of the underlying function, even if a better overall fit could be obtained otherwise. The default is `true`.
- `refine_region`: A string description of a region of the IvP domain within which further directed refinement is requested. See Section 9.1.3.
- `refine_piece`: A string description of the size and shape of uniform pieces to be used within a region of directed refinement. See Section 9.3.
- `refine_point`: A string description of a point within the IvP domain to direct further refinement. See Section 9.3.
- `smart_amount`: The number of pieces to use in the smart refinement algorithm, beyond the number of pieces used in an initial simple uniform function. See Section 9.4.
- `smart_percent`: The number of pieces to use in the smart refinement algorithm specified as a percentage of the number of pieces used in an initial simple uniform function. See Section 9.4
- `smart_thresh`: A threshold given in terms of worst noted error between the IvP function and the underlying function, below which the smart refinement algorithm will cease further refinement. See Section 9.4.
- `auto_peak`: Set to either `true` or `false` indicating whether the auto-peak algorithm should be applied. See Section 9.5.
- `auto_peak_max_pcs`: The maximum amount of new pieces added to an IvP function during the auto-peak heuristic. See Section 9.5.

`bool setParam(string param, double value):` The parameters that may be set via this function may also be set via the `setParam()` function above where the `value` parameter is a string. This alternate method is implemented solely as a convenience to the caller.

- `uniform_amount`: See above.
- `smart_amount`: See above.
- `smart_percent`: See above.
- `smart_thresh`: See above.
- `auto_peak_max_pcs`: See above.

9 Optional Advanced Features of the `Reflector` Tool

9.1 Preliminaries

The previous section discussed how to build IvP functions with the `OF_Reflector` tool in the IvP Build Toolbox by simply specifying a desired number of pieces in the resulting piecewise defined function. This section discusses a few further methods for building functions that give the user more control of the build process and typically better overall results in terms of fewer pieces, less time to build, and greater accuracy in the piecewise approximation of the underlying function.

9.1.1 The Reflector-Script

The basic invocation of the Reflector `create()` function may take a single argument requesting the number of pieces to be used in the piecewise function. An example is line 15 in Listing 25 on page 79. In reality the invocation of `create()` is comprised of a *script* of distinct build heuristics of which the creation of uniform sized pieces is just the first of four parts. The latter three parts are optional and require further user configuration before being included for execution in the script. The four parts are:

- Uniform function creation
- Directed refinement
- Smart refinement
- Auto-Peak refinement

These four heuristics are discussed in the next four sections. Uniform function creation is revisited since finer control can be used (with typically better results) if the choice of piece size and shape is not left to the heuristic that converts the requested total number of pieces into an actual uniform piece shape.

9.1.2 Specifying a Piece Shape or IvP Domain Point in String Format

Aspects of the `Reflector` tool require the specification of the shape of a piece used in a piecewise defined IvP function. The specification is comprised of the length of the piece for each of the n dimensions, i.e., decision variables. There are two ways to describe the lengths. Recall that the IvP domain for a variable is given by a low and high value, and the number of points. For example the variable x could range from 0 to 30 with 31 points, and y could range from -50 to 50 with 21 points. The first way to describe the length of a piece is by specifying the number of discrete points:

```
"discrete @ x:5,y:5"
```

A uniform function built over this domain with the above requested piece shape would have 35 pieces in a manner rendered in Figure 24.

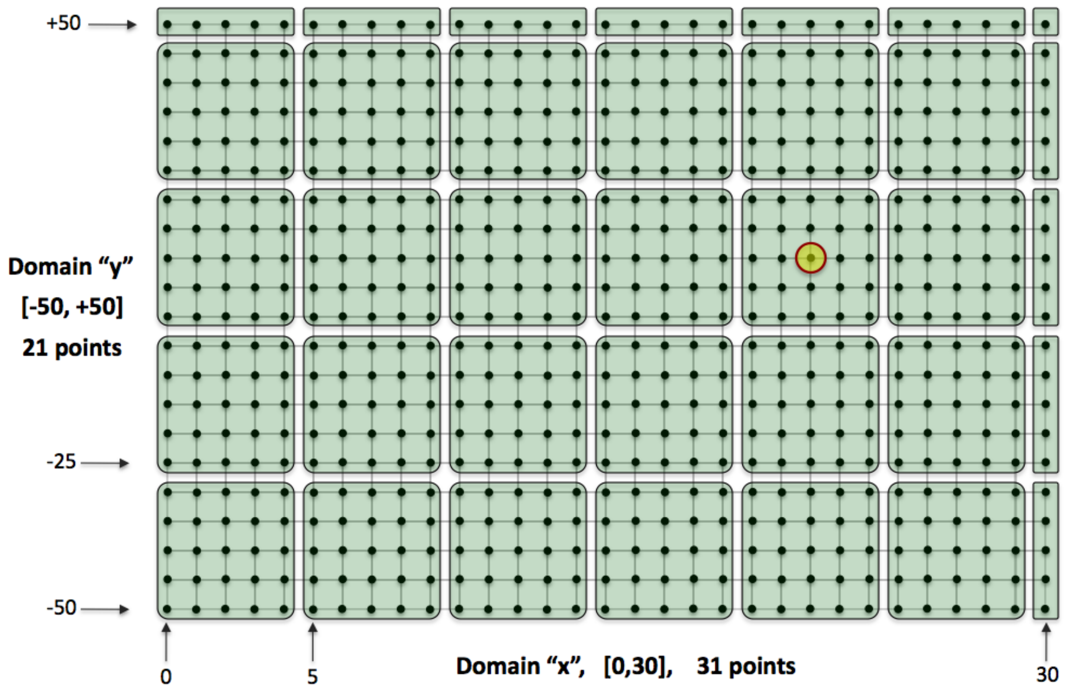


Figure 24: **A uniform IvP function**: An IvP domain is rendered over the two variables x, with 31 elements, and y with 21 elements. Requesting a set of uniform pieces with five elements on each edge results in the piece distribution shown. The circled point represents the 23rd index into the x domain and the 13th index into the y domain. This point can be referenced by the string `"discrete @ x:22,y:12"`. It may also be referenced by the string `"native @ x:22,y:10"`.

Note the distribution of pieces is not completely uniform. Smaller pieces are used at the upper ranges of the domain. A second method of specifying the same piece shape is to use the native lengths of the domain:

```
"native @ x:5,y:25"
```

This piece also has a length of five units along the x dimension and five units along the y dimension, resulting in the same distribution shown in Figure 24. When a "native" value doesn't exactly map onto one of the points in the domain, it is rounded to the nearest domain point. For example, `"native @ x:5,y:22.6"` specifies a piece with *five* units on the y dimension, `"native @ x:5,y:22.4"` specifies a piece with *four* units on the y dimension. And when a native value is given exactly between two domain points, the value is rounded up, so `"native @ x:5,y:22.5"` specifies a piece with *five* units on the y dimension.

A single *point* in the IvP domain can be similarly referenced. When the string `"discrete @ x:5,y:5"` is used to represent a piece *shape*, the numerical values represent the length of the piece. When the same string is used to represent a *point* in the IvP domain, the numerical values represent the index into the domain. For example, the circled point in Figure 24 can be be referenced by the string `"discrete @ x:22,y:12"`. It may also be referenced by the string `"native @ x:22,y:10"`. When a native values does not map exactly to a domain value, the nearest domain point is used.

9.1.3 Specifying a Region of an IvP Domain in String Format

Aspects of the `Reflector` tool require the specification of a `region` of the IvP domain. The specification is comprised of an upper and lower bound for each of the n dimensions, i.e., decision variables. Recall that the IvP domain for a variable is given by a low and high value, and the number of points. For example the variable x could range from 0 to 30 with 31 points, and y could range from -50 to 50 with 21 points. A region can be specified as follows:

```
"native @ x:10:24,y:-25:20"
```

or equivalently,

```
"discrete @ x:10:24,y:5:14"
```

This region is rendered in Figure 25. If the extents specified in the string exceed the boundaries of the IvP domain, the requested region is clipped to be exactly the boundary value. For example, the string `"native @ x:10:24, y:-25:50"` and `"native @ x:10:24, y:-25:50000"` would specify the same region given the example in Figure 25.

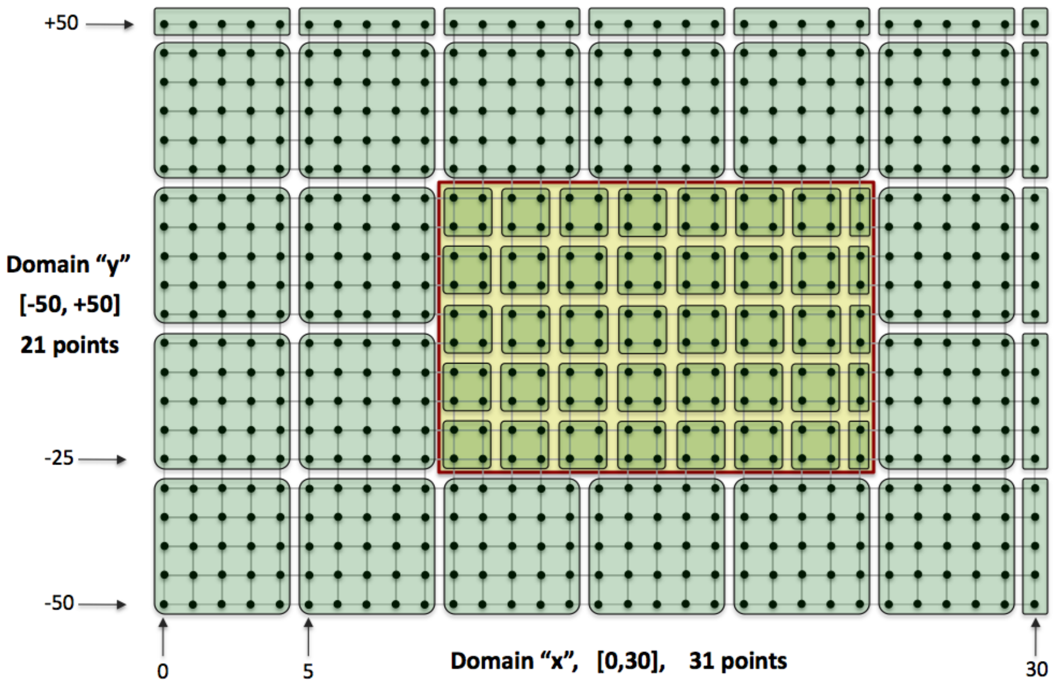


Figure 25: **A non-uniform IvP function**: An IvP domain is rendered over the two variables, x, with 31 elements, and y, with 21 elements. A region of IvP domain is identified for further application of the `Reflector`. The region is specified by the string `"discrete @ x:10:24, y:5:14"` or `"native @ x:10:24, y:-25:20"`. In this case smaller uniform pieces are applied within the region.

When a native value is specified that does not map to a domain value, this case is handled differently for *regions* than it was when specifying a piece *shape*. In a region specification the native value is treated as a strict boundary value. Therefore the string `"native @ x:9.01:24.99, y:-29.99:24.99"` would specify the exact same region as the example above and in Figure 25.

9.2 Optional Feature #1: Choosing the Piece Shape in Uniform Functions

9.2.1 Potential Advantages

By simply specifying the desired number of pieces, the `Reflector` heuristically sets the piece size and aspect ratio of an initial uniform function. This has the advantage of being very simple and independent of the underlying function. (See line 15 in Listing 25 on page 79.) However, like most heuristics, there may be cases where the result may not be best for a particular situation. If the user has some insight into the underlying function and the IvP domain, the user may not wish to leave this decision to the heuristic, but instead specify the piece shape explicitly. Below, the piece count-to-piece shape heuristic is described as well as how to override the heuristic with an explicit shape request.

9.2.2 Specifying the Piece Shape Implicitly from a Piece Count Request

When the `Reflector` creates a uniform IvP function based on a requested piece count, a heuristic is invoked to generate a single piece to be used in the uniform function based on both the piece count and the IvP domain. This piece is not unlike the 5 x 5 piece in Figure 24 on page 85, except that a 5 x 5 piece is not explicitly requested, but rather the total pieces in that figure, 35, would be requested. Knowing a little about this heuristic can help determine when its worth the effort to instead explicitly define the shape of the uniform piece. The total requested pieces is an upper limit, and often not exactly achieved. For example, the same 35 pieces in Figure 24 would be created upon piece-count requests of 35, 36, 37, 38, and 39 pieces. The heuristic attempts to keep the aspect ratio of the uniform piece close to 1.0, but will deviate to allow a uniform piece that will result in a total number of pieces closer to the requested amount. The heuristic is given Listing 26 below, and some examples are shown in Table 3.

Listing 26 - The heuristic for generating a uniform piece based on piece-count and domain.

```
IvPBox buildUniformPiece(IvPDomain domain, int max_amount)
{
  int dim = domain.getDim();
  vector<int>  pcs_on_edge(dim,1);
  vector<bool> pcs_maxed(dim,false);
  vector<int>  pts_on_edge(dim,0);

  // Store the number of points on an edge for quick reference
  for(i=0; i<dim; i++)
    pts_on_edge[i] = domain.getVarPoints();

  // Augment the number pieces on edges until done
  bool done = false;
  while(!done) {
    // Algorithm done if augmentations for all dimensions are maxed out.
    done = true;
    for(i=0; i<dim; i++)
      done = done && pcs_maxed[i]

    // Find the dimension most worthy of further augmentation
    if(!done) {
      int    augment_dim;
      double biggest = 0;
      for(d=0; d<dim; d++) {
          if(!pcs_maxed[d]) {
          double ratio = (pts_on_edge[d] / pcs_on_edge[d]);
26          if(ratio > biggest) {
27            biggest = ratio;
28            augment_dim = d;
29          }
30        }
31      }
32
33      // Augment the pieces_on_edge for the chosen dimension
34      pcs_on_edge[augment_dim]++;
35
36      // Calculate hypothetical number of boxes given new augmentation.
37      double hypothetical_total = 1;
38      for(d=0; d<dim; d++)
39        hypothetical_total *= pcs_on_edge[d];
40
41      // If max_amount exceeded, undo the augment, and max-out the dimension
42      if(hypothetical_total > max_count) {
43        pcs_maxed[ix] = true;
44        pcs_on_edge[augment_dim]--;
45      }
46
47      // Cant have more pieces on an edge than points on an edge
48      if(pcs_on_edge[augment_dim] >= pts_on_edge[augment_dim])
49        pcs_maxed[augment_dim] = true;
50    }
51  }
52
53  // Now build the uniform piece based on pts_on_edge and pcs_on_edge
54  IvPBox uniform_piece(dim);
55  for(d=0; d<dim; d++) {
56    double edge_size = ceil(pts_on_edge[d] / pcs_on_edge[d]);
57    uniform_piece.setPTS(d, 0, edge_size-1);
58  }
59  return(uniform_piece);
60 }
```

The heuristic progresses by growing the number of "pieces on an edge", `pcs_on_edge`, on each dimension. The algorithm proceeds to grow the `pcs_on_edge` for each dimension until it cannot grow further. For example, in Figure 24 there are seven pieces on the x edge and five pieces on the y edge. The algorithm is initiated with a single piece on each edge, i.e., dimension, (line 3 in Listing 26). A Boolean is associated with each dimension indicating whether growth in that dimension has been maxed out. This vector is initiated on line 4. A dimension becomes maxed out if additional growth in that dimension means the requested piece count is exceeded (checked for in lines 36-45), or if the number of pieces on an edge is equal to the number of points on and edge of the IvP domain (checked for in lines 47-49). At each chance to grow the size of the uniform piece the most appropriate dimension is identified for growth (lines 21-31) by choosing the dimension with the largest ration of points on the edge to pieces on the edge (line 25).

Some examples of the heuristic are shown in Table 3. The domain shown in table has 1000 discrete choices for both the x and y variables. Given that the domain itself has an aspect ratio of one, not surprisingly, the generated uniform pieces also have roughly an aspect ratio of 1.0, and the number of pieces on each edge of the domain are also nearly equivalent.

Requested Pieces	Aspect Ratio	Shape of Piece	Actual Pieces	Pieces On the 'x' Domain Edge	Pieces On the 'y' Domain Edge
63	0.78	112x143	63	9	7
64	1.0	125x125	64	8	8
500	1.0	46x46	484	22	22
512	0.96	44x46	506	23	22
1000	0.97	32x33	992	32	31
1024	1.0	32x32	1024	32	32
1025	1.0	32x32	1024	32	32
4000	1.0	16x16	3969	63	63
IvPDomain: x:200:299:1000, y:0:999:1000					

Table 3: **Example 2D results of the uniform-piece heuristic**: Uniform piece characteristics resulting from a heuristic applied to a requested total number of pieces and a given IvP domain with two variables.

Consider how the heuristic performs instead on the 3D domain shown in Table 4. The number of choices for the z variable is a tenth of that for the x and y variables. The results provided by the heuristic may or may not be the right overall, depending on the underlying function and application. In particular consider that when requesting 100 or 200 pieces, the z component of the resulting uniform piece is the entire z domain, i.e., there is only one piece on the z domain edge.

Requested Pieces	Shape of Piece	Actual Pieces	Pieces On the 'x' Domain Edge	Pieces On the 'y' Domain Edge	Pieces On the 'z' Domain Edge
100	100x100x100	100	10	10	1
200	72x72x100	196	14	14	1
1000	44x46x50	968	22	22	2
7500	23x24x25	7392	44	42	4
IvPDomain: x:200:299:1000, y:0:999:1000, z:0:99:100					

Table 4: **Example 3D results of the uniform-piece heuristic**: Uniform piece characteristics resulting from a heuristic applied to a requested total number of pieces and a given IvP domain with three variables.

This heuristic has served fairly well in practice, but in cases where the user has insight into a better choice for the size and shape of the uniform piece, this can be overridden as discussed next.

9.2.3 Specifying the Uniform Piece Shape Explicitly

The piece shape used in a uniform IvP function can be set explicitly using the `uniform_piece` parameter in the Reflector `setParam()` function first mentioned in Section 8.4 on page 82. For example, the uniform piece shown in Figure 26 can be requested as follows:

```
reflector.setParam("uniform_piece", "discrete @ x:6,y:4");
int amt = reflector.create();
```

Compared to generating a uniform function by a simple piece-count request, the above two lines would replace the single line with the `create()` invocation, as in line 15 in Listing 25 on page 79.

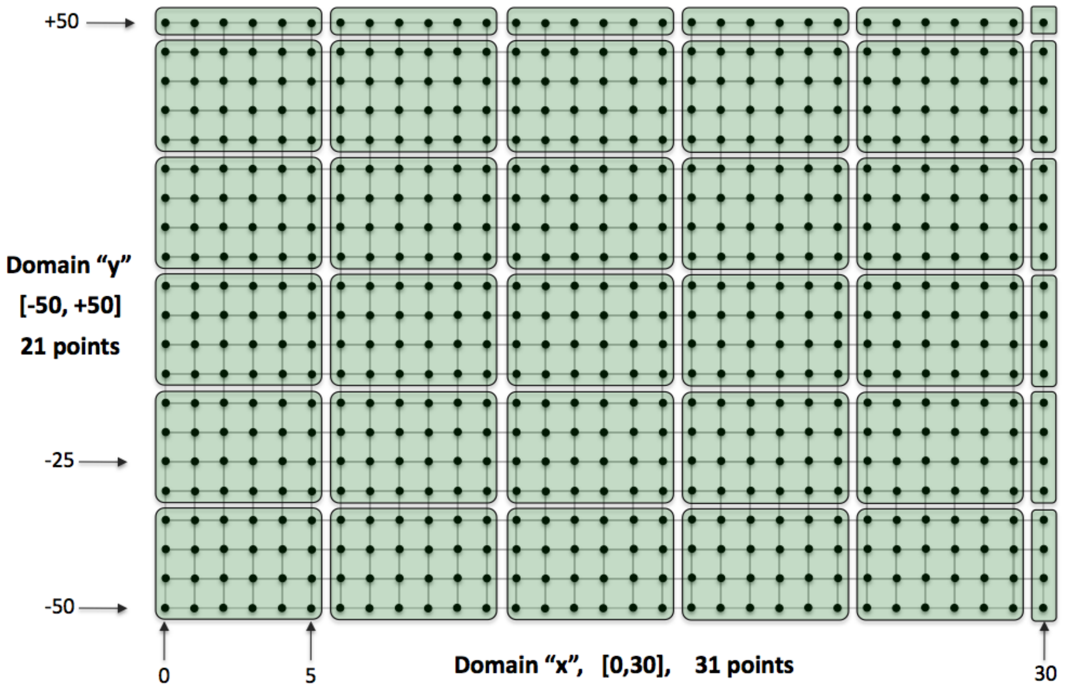


Figure 26: **An IvP function made from an explicit piece shape request**: An IvP domain is rendered over the two variables x, with 31 elements, and y with 21 elements. Requesting a uniform piece of size 6x4 would result in the rendered configuration. This piece can be specified with `"discrete @ x:6,y:4"` or `"native @ x:6,y:20"`. This piece shape would *not* be resulting piece shape had the user simply requested 36 pieces given this domain.

In summary, when the Reflector `create()` function is called, the reflector-script begins and needs to know the size and shape of the piece used for uniform function creation. It may get this information by either explicitly configuring the piece shape, or implicitly by requesting a total number of pieces (as an argument to the `create()` function). If both requests are inadvertently invoked, the latter type of request is ignored and the explicit piece shape configuration is honored. If neither specification of piece shape is provided, a function with a single piece will be created (but perhaps further refined in later parts of the reflector-script). Use of the explicit piece shape request may be the preferred method for example if a domain includes a variable for vehicle heading and a uniform function is desired with pieces split on every three degrees, regardless of whether the domain contains 180, 360, or 720 choices for heading.

9.3 Optional Feature #2: IvP Functions with Directed Refinement

The directed-refinement feature of the Reflector is potentially useful when (a) the underlying function has distinct sub-regions that are harder to accurately represent with a piecewise linear approximation, and (b) when the user has insight into the location of those sub-regions. Use of the tool involves specifying both the region to direct further refinement, and the size of the piece to use in the refinement region. This is done using the `uniform_piece`, `refine_region`, and `refine_piece` parameters in the Reflector `setParam()` function first mentioned in Section 8.4 on page 82. For example, the IvP function shown in Figure 25 would be generated with the following lines:

Listing 27 - An example configuration of the `Reflector` *tool using directed refinement.*

```
reflector.setParam("uniform_piece", "discrete @ x:5,y:5");
reflector.setParam("refine_region", "native @ x:10:24,y:-25:20");
reflector.setParam("refine_piece",  "discrete @ x:2,y:2");
reflector.create();
```

When the `create()` function is invoked in the last line above, the reflector-script will involve two of the components of the reflector-script mentioned in Section 9.1.1. The first line configures the initial uniform function phase, and the middle two lines configure the directed refinement phase by declaring a sub-region (second line) and a uniform piece to be applied to that sub-region (third line). Multiple directed refinements can be configured and queued for inclusion in the reflector-script by adding further `refine_region` - `refine_piece` pairs prior to the invocation of the `create()` function. They must be added in pairs however since the `refine_piece` is always associated with the last specified `refine_region`.

For an illustrative case we return to the Gaussian function rendered in Figure 23 on page 81:

$$f(x,y) = Ae^{-(\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})}, \tag{2}$$

where $A = 150$, $x_0 = 50$, $y_0 = -150$ and $\sigma = 32.4$. This function apparently has a sub-region of the domain where the function is very nonlinear and otherwise quite linear outside the sub-region. The use of directed-refinement begins by building an initial uniform function as shown in Figure 27, conceding for now that the approximation will be poor in the sub-region around the peak.

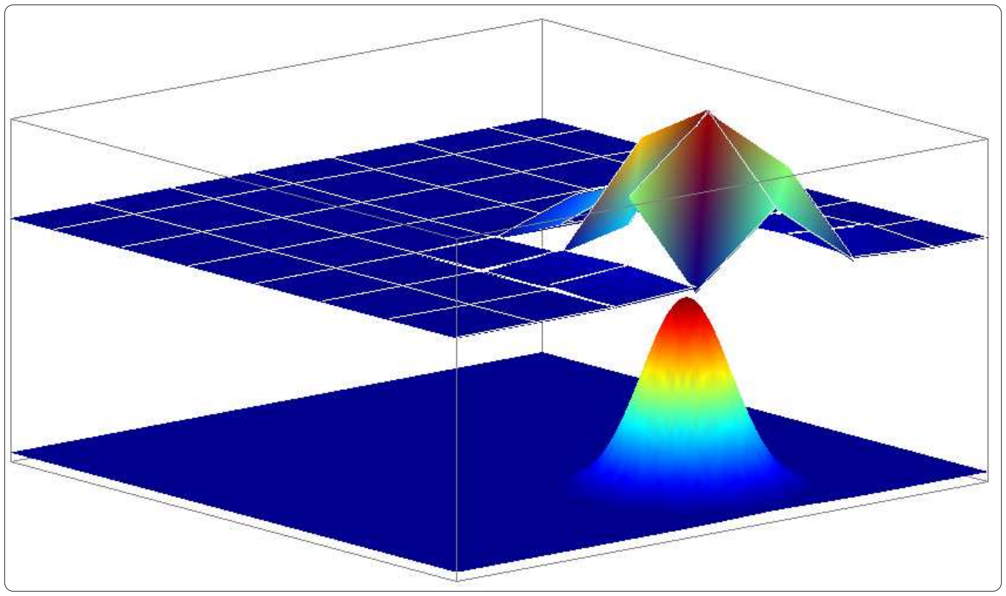


Figure 27: **An initial IvP function approximation**: The `Reflector` first creates an initial simple uniform function with fairly large pieces, conceding for the time being poor performance in approximating the underlying function in areas near the peak of the underlying function.

The initial uniform function was created by requesting 50 pieces, and a function with 49 pieces was subsequently generated. The sub-region shown in Figure 28 was identified for directed refinement, with much smaller pieces used in the sub-region.

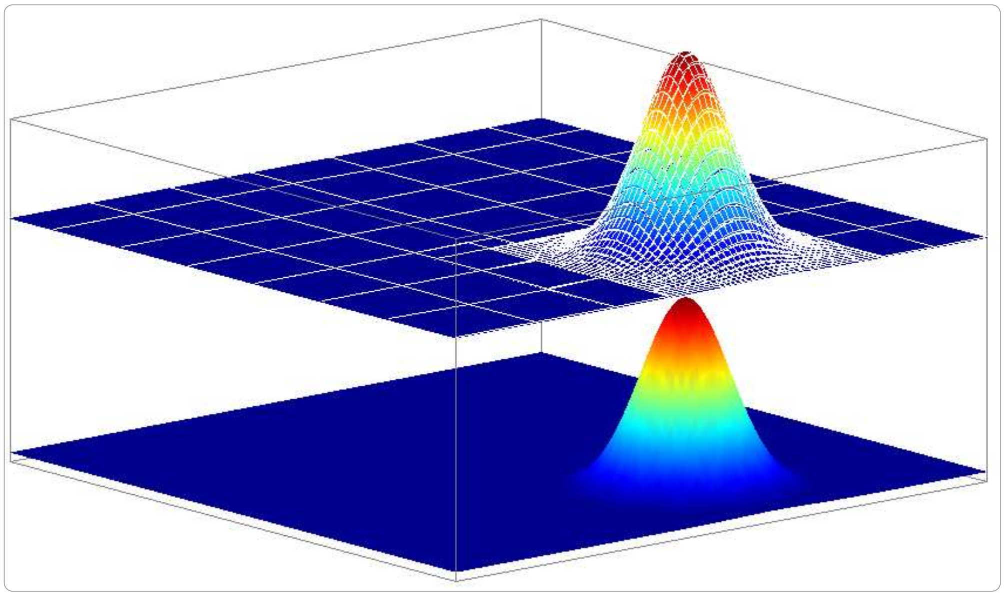


Figure 28: **An IvP function generated with directed-refinement**: After an initial uniform function has been generated, the `Reflector` refines the function on the prescribed sub-domain of the function with much smaller pieces.

The results in Table 5 below were generated by configuring the reflector-script to include directed-refinement on the underlying Gaussian function shown in Figure 28, in a manner similar to the four lines in Listing 27 on page 91. Each row in the table below differs only in the size of the `refine_piece` shown in the second column.

Case	Refine Edge Size	Total Pieces	Worst Error	Average Error	Time millisecs
A	4	2607	0.7760	0.0104	38.9
B	5	1687	0.7760	0.0123	25.6
C	6	1162	0.7760	0.0149	17.8
D	7	847	0.7760	0.0178	13.0
E	8	682	0.9093	0.0214	10.2
F	9	535	1.1895	0.0254	8.2
G	10	447	1.4589	0.0302	6.7
H	11	367	1.8263	0.0351	5.7
I	12	295	2.2470	0.0409	4.7
J	13	262	2.6527	0.0472	4.1
K	14	231	3.0321	0.0540	3.6
L	15	202	3.5886	0.0615	3.2

Table 5: **Results of directed-refinement**: Characteristics of 12 different IvP functions approximating the underlying function shown in Figure 27 and 28. Each function is built by starting with an initial uniform function and then performing directed refinement over the region $-50 \leq x \leq 150$ by $-250 \leq y \leq -50$. The refinement piece size shown in the second column is the parameter that results in the 12 different functions.

For the observed errors reported in columns 4 and 5, the domain was sampled for each resulting IvP function at 50,000 random points for comparison between the value provided by the IvP function against the underlying function. The average time to create the IvP function, noted in the last column, was taken by averaging 100 creations since the precision of the timer used was 100 milliseconds. The data shown here are meant to show the relationship between parameters, not necessarily an indication of how fast things run on "typical" platforms. That being said, this data is from a Dell laptop containing a Pentium chip with about 2.0 GHz processor, with a codebase compiled without typical gcc optimization options.

The trends in the table are as one would expect. As the number of pieces is decreased, the average error and worst error increase, and the time to create the IvP function is decreased. The question is whether this technique offers the ability to improve in all three metrics, piece-count, function accuracy, and creation time, simultaneously compared to using a simple uniform function without directed refinement. The answer is yes. Evidence can be seen of this by comparing Table 5 with Table 2 on page 80. We look for cases in Table 5 that dominate cases in Table 2. A case that dominates another is stronger or equal in all three performance metrics simultaneously. Case (a) dominates case (4). Case (b) dominates cases (5),(4). Case (c) dominates cases (6),(5). Case (d) dominates cases (6),(5). Case (e) dominates cases (7),(6).

9.4 Optional Feature #3: IvP Functions with Smart Refinement

9.4.1 Potential Advantages

The smart-refinement algorithm works by further refining an existing IvP function based on an (automated) estimate of which pieces need refinement the most. There are two key ideas in this algorithm. First, no insight into the underlying function form is required by the user, unlike the directed-refinement tool. Second, the prioritization of pieces is based on the apparent fit between a piece's linear function and the underlying function, for the sub-domain of that piece. This determination of fit can be measured by performing very little extra computations beyond the already required calculations performed during linear regression for each piece during the uniform and directed-refinement phases. In short, there is typically very little reason not to invoke this tool to some degree.

9.4.2 The Smart-Refinement Algorithm

The smart-refinement algorithm utilizes information collected during the creation of pieces earlier in the reflector-script, during the initial uniform function phase which is always invoked, and directed-refinement phase which is optionally invoked. During these phases, pieces are formed and linear regression is performed to determine the linear function associated with each piece.

To perform linear regression for a new piece, the underlying function over k variables is sampled at $n = 2k + 1$ points (the corners and the middle point) to produce $f(\vec{x}_1) \ldots f(\vec{x}_n)$, and these n values are used to determine the linear function for that piece:

$$f'(\vec{x}) = c_1 x_1 + \ldots + c_k x_k + b. \tag{3}$$

The same n points are again evaluated using the newly determined linear function (3) instead, producing another set of n values $f'(\vec{x}_1)\ldots f'(\vec{x}_n)$. The `regression score` is determined by:

$$\texttt{regression\_score} = \text{sqrt} \sum_{i=1}^{n}(f'(\vec{x}_i) - f(\vec{x}_i))^2 \tag{4}$$

The `regression_score` is then inserted into a priority queue along with a reference to the piece that generated the score. The idea is shown in Figure 29. This algorithm for implementing a priority queue can be found in [8]. The priority queue implemented in the IvPBuild Toolbox is modified slightly to be a fixed-length queue. Insertion and retrieval time is $O(n\ log(n))$.

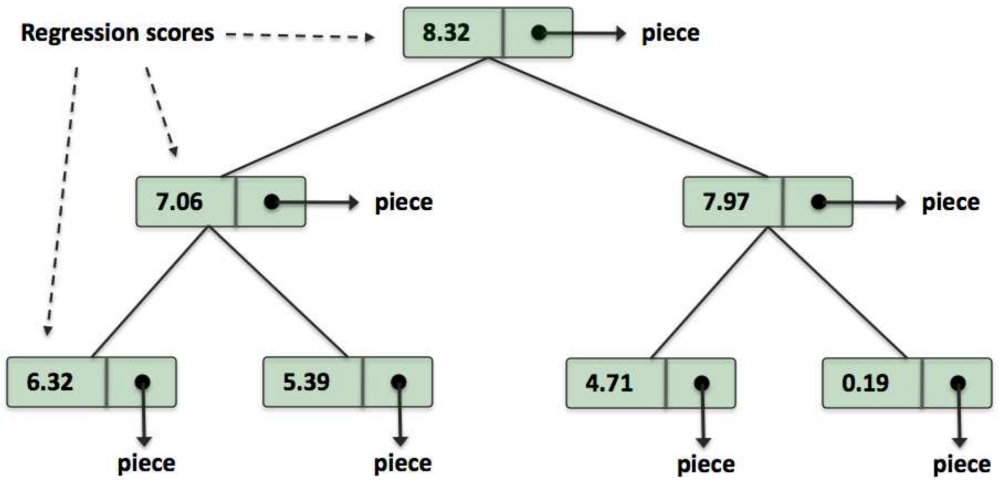


Figure 29: **Priority queue keyed with regression scores**: The `Reflector` uses a balanced priority queue based on a regression score to determine which pieces could benefit the most from further refinement.

The `Reflector` instance maintains this priority queue only if smart-refinement is activated. The pieces made during the initial uniform function and directed-refinement parts of the reflector-script are stored in the priority queue. The smart-refinement proceeds by repeatedly popping the top priority piece from the queue for further refinement. By further refinement of a piece, we mean splitting a piece and replacing the piece with the two new pieces after performing regression on the two new pieces. The piece is split along the dimension with the largest edge. These two new pieces are then also inserted in the priority queue for possible further refinement.

An example of the smart-refinement algorithm applied to the same Gaussian function shown in Figure 28 on page 92 is shown below in Figure 30.

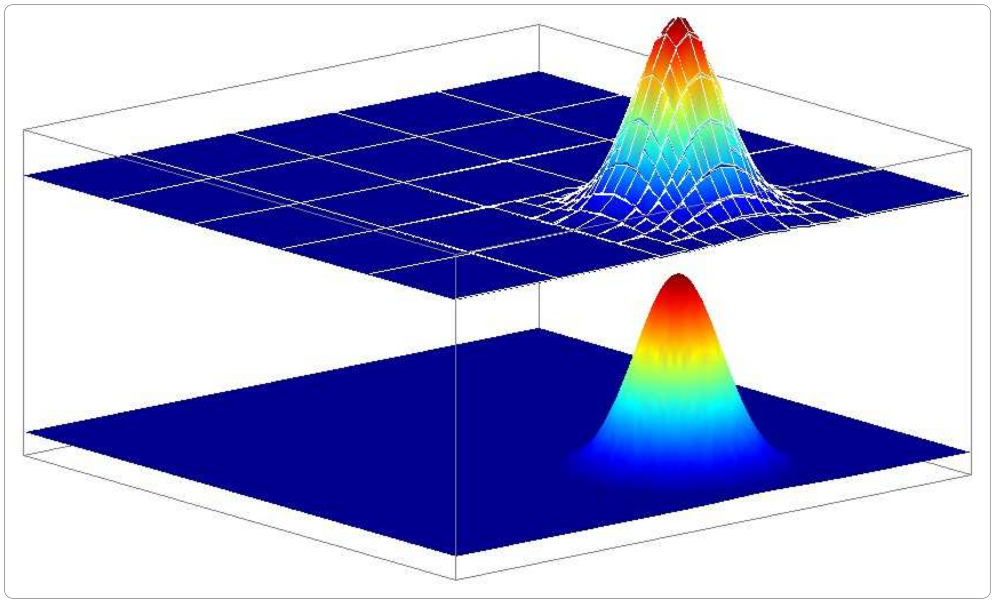


Figure 30: **An IvP function generated with smart-refinement**: Results of smart-refinement on an initial uniform function with 25 pieces and an additional 200 pieces during the smart-refinement phase. The function has a total of 225 pieces and is significantly more accurate and faster to create than a pure uniform function with 625 pieces. Further examples are shown in Table 6.

The results in Table 6 show the results of applying the smart-refinement algorithm to the function in Figure 22. Each row in the table shows the results from creating first a pure uniform function, and then a further refined function using an additional 75% more pieces with smart-refinement. The left-hand side of the table is the same as Table 2, duplicated here for ease of comparison. Compare for example the smart-refine function with 175 pieces in Case 10 against the pure uniform function with 400 pieces in Case 7. The former not only has less pieces, but is more accurate and took less time to create. It dominates the pure uniform function, i.e., is simultaneously better in all measures of performance. This is similar to the way directed-refinement dominates pure uniform functions, but in the case of smart-refinement, no insight into the underlying function form was required!

Case	Edge Size	Pieces	Worst Error	Avg Error	Time msec	Pieces	Worst Error	Avg Error	Time msec
4	10	2500	1.4589	0.0232	39.9	3445	0.8512	0.0139	70.1
5	15	1156	3.4532	0.0551	18.9	2023	1.3241	0.0224	47.3
6	20	625	5.5855	0.1014	10.4	1093	1.9834	0.0297	27.6
7	25	400	7.79764	0.1585	6.5	700	2.5924	0.0362	18.1
8	30	289	12.0347	0.2303	4.7	505	2.3905	0.0480	11.3
9	40	169	24.2977	0.3919	2.8	295	12.6192	0.0885	7.5
10	50	100	18.2113	0.5917	1.6	175	3.4166	0.1194	4.3
11	75	49	42.0652	1.2143	0.9	85	7.9236	0.3100	2.3
12	100	25	30.3938	2.0285	0.5	43	12.3887	0.5188	1.2

Table 6: In each case an initial uniform function was created with the number of pieces indicated in column 3. The qualities of the function in terms of accuracy and time are shown in columns 4-6. Each function was then augmented with 75% additional pieces using the smart-refinement algorithm with the resulting qualities shown in columns 7-10.

The smart-refine algorithm is limited by the degree to which the regression score given by (4) is accurate for each piece entered into the queue. Note for example, Case 9 in Table 6, where the worst error detected for the smart-refine function is anomalous and significantly higher than that noted in functions with far fewer pieces. The error of 12.6192 occurred in some piece that apparently did not report a high regression score when (4) was applied. This is likely due to an unfortunate case where the points sampled for use in generating the linear function all fit the resulting linear function very well, but the non-sampled points did not fit well. The idea is shown in Figure 31.

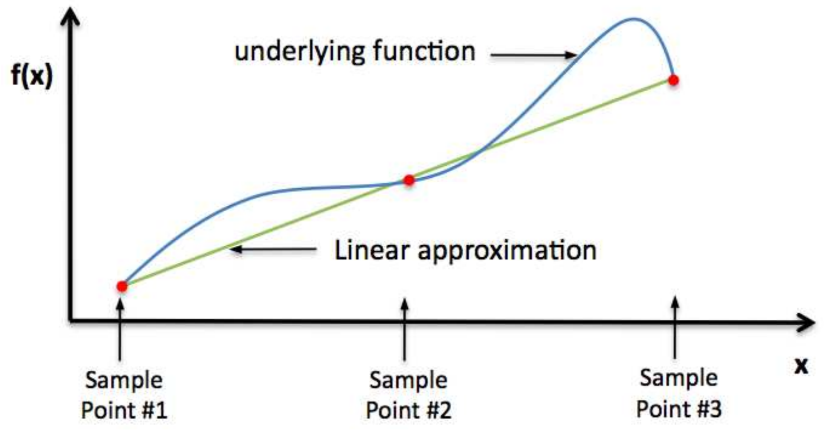


Figure 31: **Regression scoring gone awry**: Assessing the regression score given in (4) can be misleading in cases where the derived linear function fits each sampled point very well but otherwise poorly fits the underlying function.

9.4.3 Invoking the Smart-Refine Algorithm in the Reflector

The smart-refine tool can be included in the refine-script in a few different ways. The first is to simply indicate how many pieces to use during the smart-refine process. The number of pieces is addition to any pieces that may already be present after the initial uniform function has bee created and after any directed-refinement has been performed.

```
reflector.setParam(smart_amount", 400);
```

Alternatively the number of pieces to be used in smart-refine can be given in terms of the percentage of additional pieces beyond what has already been created at the time of invocation of the smart-refine part of the refine-script. The argument is a non-negative integer value:

```
reflector.setParam("smart_percent", 35);
```

There is a third parameter *smart_thresh* that can affect how many pieces are used in the smart-refine phase. The value passed with this parameter is a regression score such that if the current top element of the priority queue has a score below this threshold, smart-refinement will terminate early, before the target piece amount specified by *smart_amount* or *smart_percent* has been reached:

```
reflector.setParam("smart_thresh", 0.05);
```

Regression scores represent the raw discrepancy between the underlying function and the linear approximation (3), and in general do not reflect any normalization. For example, depending on the function, the value of 0.05 above could be a relatively large value resulting in an early termination of refinement, or a relatively small threshold that cannot be met without thousands of additional pieces. The user of this parameter needs to have some knowledge of the range of the underlying function.

Finally, the simplest way of invoking the smart-refinement tool is by specifying the number of pieces as the second argument in the `create()` function call, and the smart threshold as the third argument:

```
reflector.create(1000, 400, 0.5);
```

The above will result in an initial uniform function with 1000 pieces and an additional 400 pieces used for smart-refinement. The full additional 400 pieces will be generated only if the threshold is not reached along the way. The above is equivalent to:

```
reflector.setParam("uniform_amount", 1000)
reflector.setParam("smart_amount", 400)
reflector.setParam("smart_thresh", 0.5)
reflector.create();
```

Accepting these three common parameters as arguments to the `create()` function call is simply for convenience. If provided, they override the setting from prior calls to `setParam()`.

9.5 Optional Feature #4: IvP Functions with Auto-Peak Refinement

9.5.1 Potential Advantages

The auto-peak algorithm is the last optional algorithm in the reflector-script. The objective is also to build a more accurate IvP function representing the underlying function. The metric of accuracy referenced up to this point has been the average error and worst error observed from a number of random sample points. For example Table 6 on page 95 reported error in this way. Another metric of accuracy is the degree to which the maximum peak of the function, represented by a point in the discrete IvP domain, agree between the underlying function and the IvP function. For example, if the peak of the underlying function is `"heading=133, speed=3.2"` and the peak of the IvP function is `"heading=129, speed=3.0"` the IvP function could still be rated well in terms of error metrics from sampling the entire domain. However, the peak of the function is probably the most important part of the underlying function to represent precisely. When the IvP function happens to be the only function or dominating function influencing the vehicle at that moment, the peak of the function *is* the output of the solver. The auto-peak refinement focuses on this aspect of the function, without user insight into where the actual peak occurs in the underlying function.

9.5.2 The Auto-Peak Algorithm

The auto-peak algorithm proceeds by repeatedly refining the single piece in the IvP function that is believed to contain the maximum peak until that one piece contains only a single point in the IvP domain. It takes advantage of the fact that for a given piece in an IvP function, its maximum value, over the piece interval and linear interior function, can be rapidly calculated. The basic algorithm steps are as follows:

Listing 28 - An overview of the auto-peak algorithm.

```
Step 1.  Determine max-value for each piece in the IvP function and populate the priority queue.
Step 2   If the top piece in the priority queue contains only one IvP domain point, go to 9.
Step 3   If the number of pieces added during auto-peak has reached an upper limit, go to 9.
Step 4.  Pop the top piece in the priority queue for further refinement.
Step 5.      Split the piece along the longest edge.
Step 6.      Build the new linear function for both pieces, noting max values.
Step 7.      Add both new pieces back to the IvP function and to the queue with their max-values.
Step 8.      Go to Step 2.
Step 9.  Done.
```

The first step in the algorithm is to build a priority queue similar to the priority queue used in the smart-refinement algorithm. In this case, the score associate with each piece in the queue is the maximum value for the given piece, as depicted in Figure 32 below. The maximum value is calculated quickly and directly from the coefficients of the linear function associated with the piece. When the auto-peak algorithm is initiated, it works with the IvP function generated thus far during the prior phases of the reflector-script. The priority queue is built by evaluating the max-value for all pieces in this function.

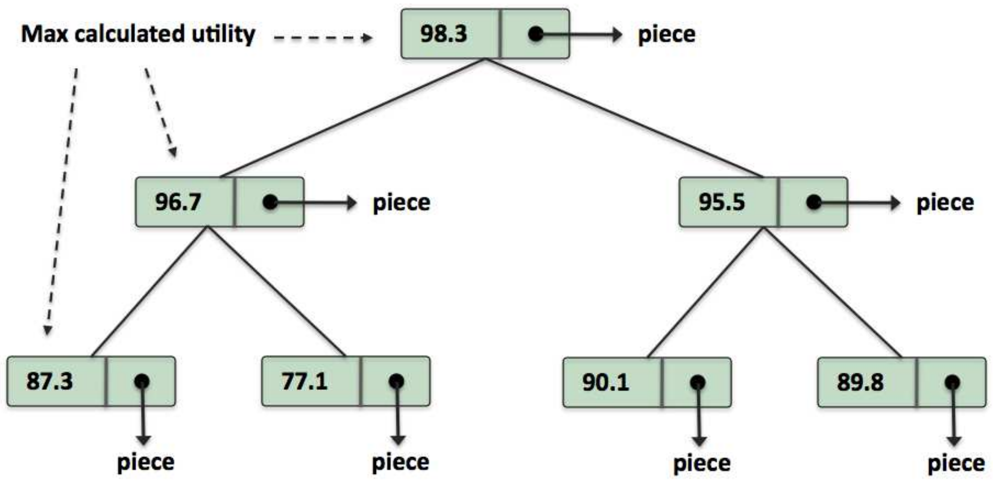


Figure 32: **Priority queue keyed with maximum utility scores**: The Reflector uses a balanced priority queue based on a max utility score to determine which pieces could benefit the most from further refinement. Each node in the tree keeps a pointer to the piece that generated the maximum utility key.

The algorithm terminates when either the top piece in the priority queue contains only a single IvP domain point, or when auto-peak refinement has generated a total of new pieces that exceeds a specified optional limit. This is checked in steps 2-3 in Listing 28. When a piece is selected for further refinement, it is split along the longest edge creating two pieces (one new piece) regardless of the number of edges or dimensions. Linear regression is performed on the two pieces and they are added back to the IvP function and the priority queue. Typically, but not necessarily, the piece with the maximum value in the priority queue is a result of the most recent refinement.

9.5.3 Invoking the Auto-Peak Algorithm in the Reflector

Use of the auto-peak tool is done using the `auto_peak` and `auto_peak_max_pcs` parameters in the Reflector `setParam()` function first mentioned in Section 8.4 on page 82. The `auto_peak` parameter simply turns the tool on or off, with `"true"` or `"false"`. The `auto_peak_max_pcs` parameter sets an upper limit on the number of new pieces introduced to an IvP function during the auto-peak phase. The following shows an example usage:

```
reflector.setParam("auto_peak", "true")
reflector.setParam("auto_peak_max_pcs", 100)
reflector.create(1000);
```

The upper limit on pieces is typically not needed, and the default value is no limit. The algorithm tends to reach termination quickly because the piece with the maximum point tends to always be at the top of the priority queue, and in cases where the top ranked piece does not contain the maximum point, this is resolved quickly as the piece is split. Nevertheless, this upper limit is available for the conservative user. It is worth noting that, for underlying functions where the maximum value is part of a large plateau, the auto-peak tool is likely to have little benefit.

References

[1] http://www.pmel.noaa.gov/vents/acoustics/tutorial/11-sofar.html.

[2] http://en.wikipedia.org/wiki/SOFAR_channel.

[3] 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.

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

[5] 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.

[6] 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. Technical Report MIT-CSAIL-TR-2009-028, MIT Computer Science and Artificial Intelligence Lab, April 2009.

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

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, September 2001.

[9] Kalyanmoy Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, 2001.

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

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

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

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

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

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

[16] Paolo Pirjanian and Henrik I. Christensen. Behavior Coordination using Multiple-Objective Decision Making. In *SPIE Conference on Intelligent Systems and Advanced Manufacturing*, Pittsburgh, Pennsylvania, October 1997.

[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 and Charles E. Thorpe. Combining Multiple Goals in a Behavior-Based Architecture. In *International Conference on Integrated Robots and Systems (IROS)*, 1995.

Index