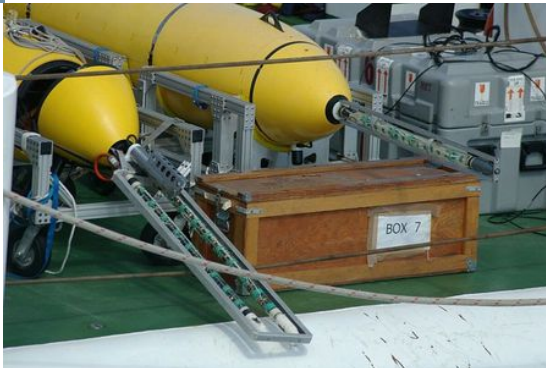




# The IvP Helm and New Features of MOOS-IvP 4.2



Michael R. Benjamin

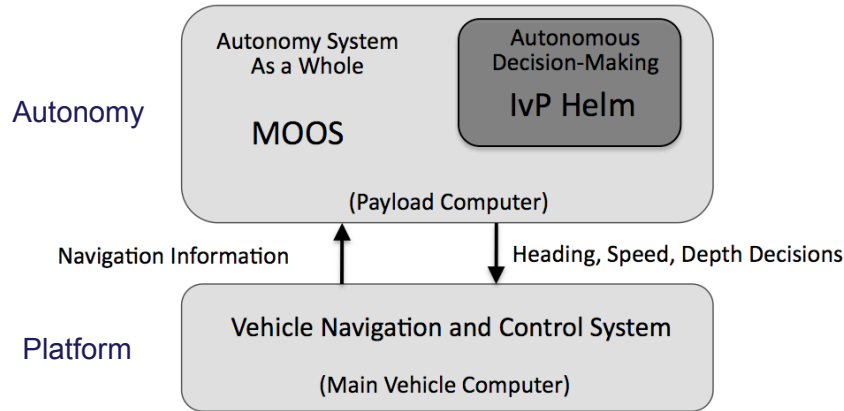
MIT Dept. of Mechanical Engineering  
Computer Science and AI Lab (CSAIL)

[mikerb@mit.edu](mailto:mikerb@mit.edu)  
<http://oceanai.mit.edu/mikerb>



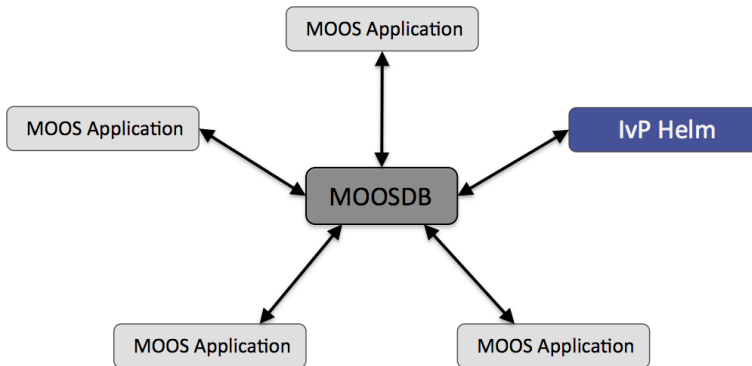
# Payload UUV Autonomy (3 Architecture Principles)

## Principle #1 – Separation of Vehicle Autonomy from the Physical Platform

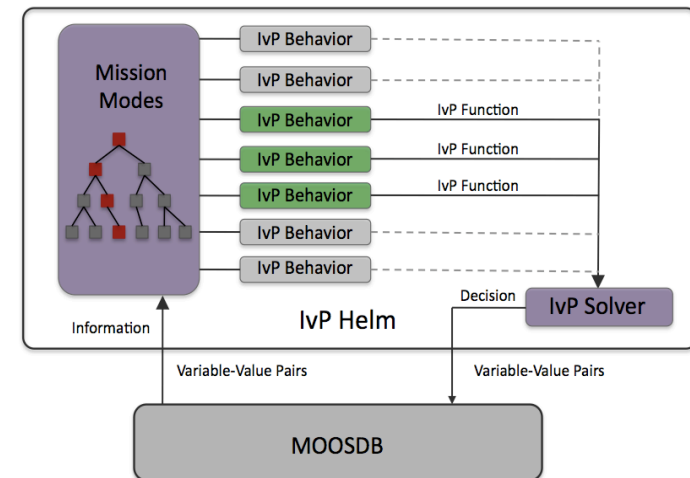


- Each architecture principle is designed to provide the end user with choices and the ability to procure capabilities on a component rather than system basis.
- Allows for incremental development across organizations.
- These new thrusts will result in tangible new software modules and capabilities.

## Principle #2 – Separation of Autonomy System Components (MOOS Middleware)



## Architecture Principle #3 – Separation of Autonomy into dedicated behaviors (IvP Helm)

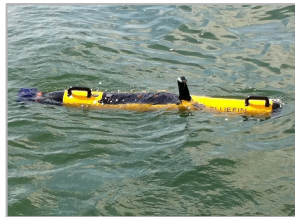


# Overview of MOOS-IvP Payload Autonomy Software

## Autonomy Software at MIT – Laboratory for Autonomous Marine Sensing Systems

- Open Source software project – 70+ applications 130,000+ lines of code, 25-30 work years of research & development.
- MOOS-IvP autonomy software - [www.moos-ivp.org](http://www.moos-ivp.org)
- Goby underwater communications software - [www.gobysoft.org](http://www.gobysoft.org)
- "An Overview of MOOS-IvP and a Brief Users Guide to the IvP Helm Autonomy Software" <http://dspace.mit.edu/handle/1721.1/45569>
- "Extending a MOOS-IvP Autonomy System and Users Guide to the IvPBuild Toolbox" <http://dspace.mit.edu/handle/1721.1/46361>
- "MOOS-IvP Autonomy Tools Users Manual" <http://dspace.mit.edu/handle/1721.1/43708>

## Platforms that have field-demonstrated MOOS-IvP payload autonomy:



Bluefin-9



Bluefin-21



REMUS 600



REMUS 100



Ocean Explorer



RMS/MIT Kayaks



Iver-2



SeaRobotics USV



H-Scientific USV



Kingfisher USV



Yellowfin AUV

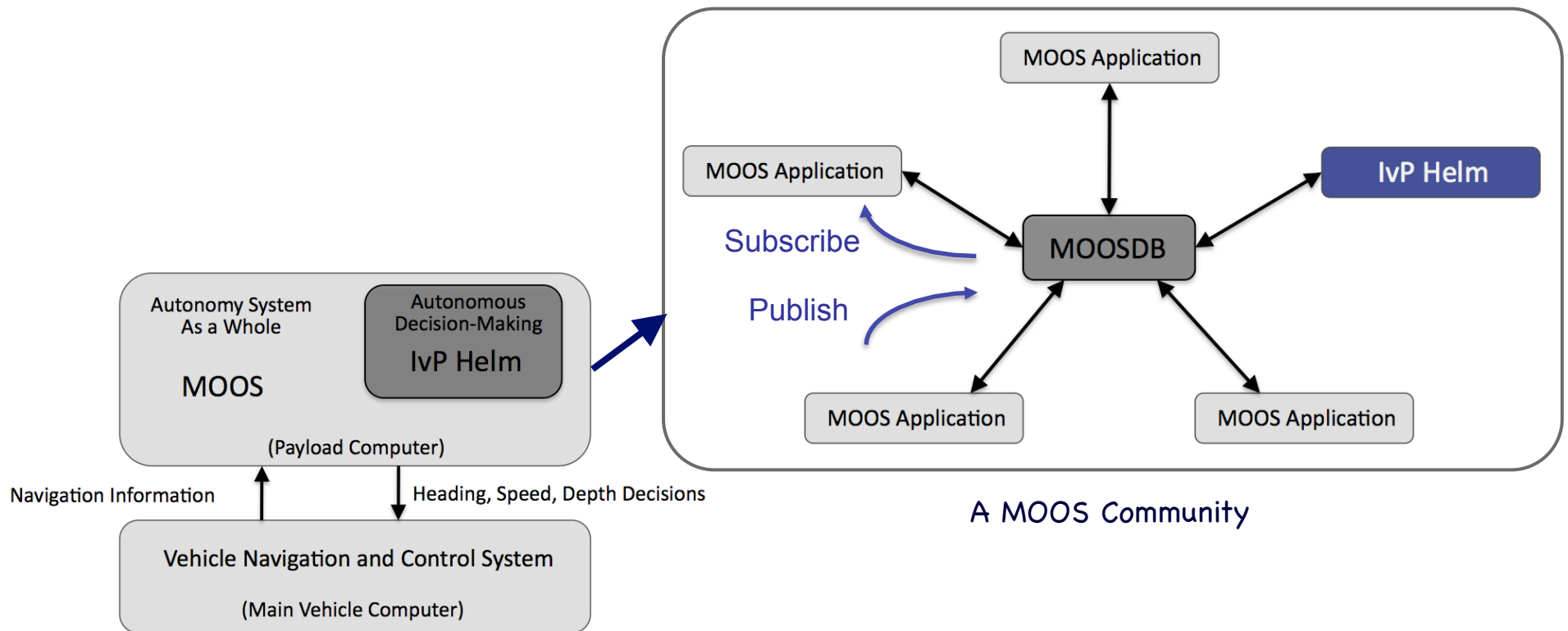


SARA USV

# The Payload Autonomy Paradigm

## Principle #2 - Separation of Autonomy System Components (MOOS)

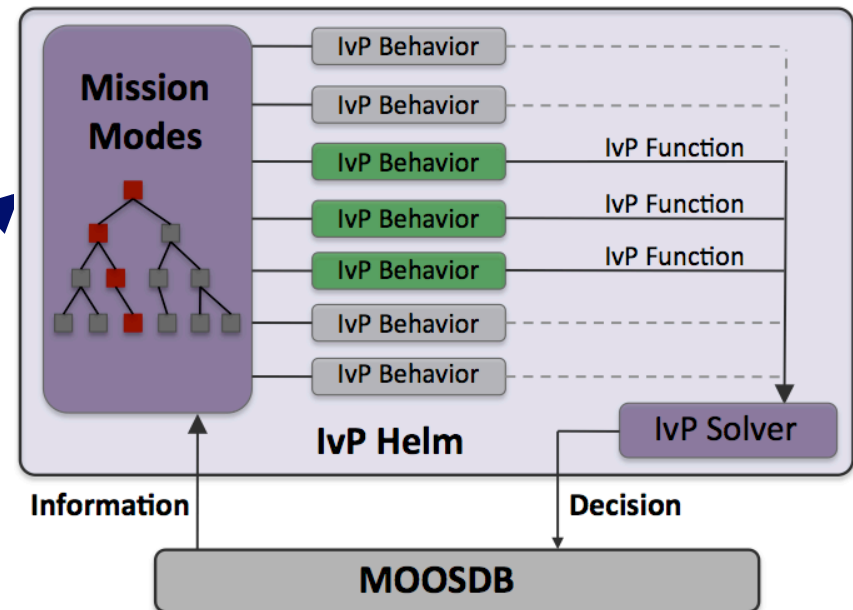
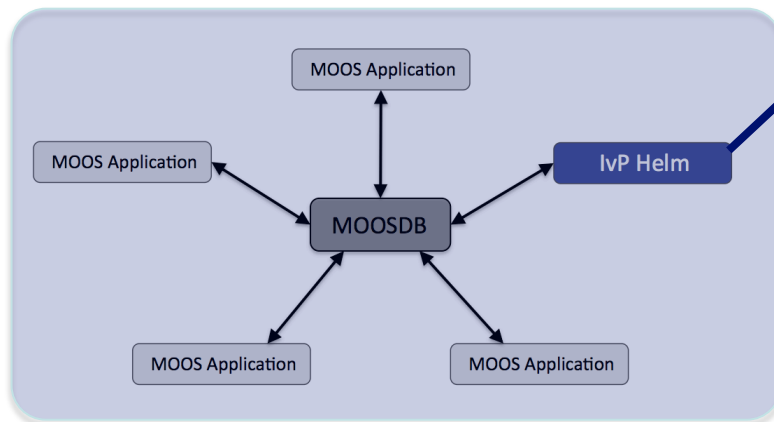
- MOOS is middleware built on the publish-subscribe architecture.
- Each MOOS application is a separate process running on the vehicle computer.
- The interface of each process is defined by the messages it publishes and the messages it subscribes for.



## Principle #3 - Separation of Autonomy into dedicated distinct behaviors.

- The IvP Helm is a decision-making engine based on the behavior-based architecture. It is a single MOOS application comprised of multiple specialized behaviors.
- Behaviors are turned on or off based on defined situations (states) and transitions. When multiple behaviors are active, coordination is by multi-objective optimization.
- Interval Programming (IvP) is the technique used for multi-objective optimization.

MOOS-IvP Payload Autonomy System



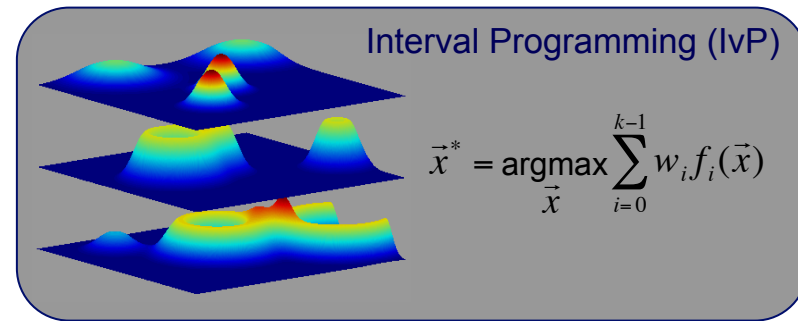
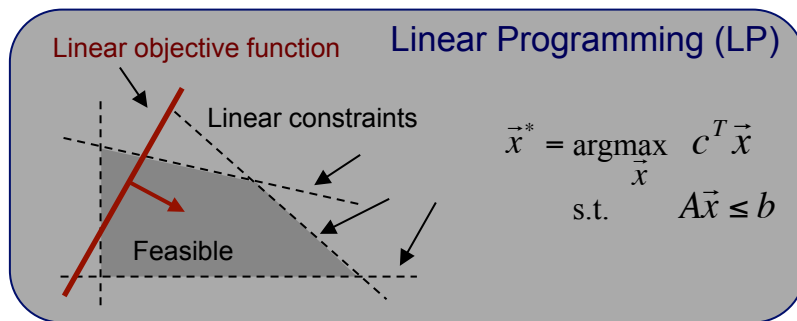
# Outline

- Trends in autonomous marine vehicles
- The Payload Autonomy Paradigm and the MOOS-IvP project
- Multi-Objective Optimization with Interval Programming
- The IvP Helm
- MOOS-IvP 4.2 and Plans for Future Development

# Interval Programming

Interval Programming is a mathematical programming model.

- A mathematical structure or syntax for representing an optimization problem.
- A toolbox of practical methods for casting different problems into the IvP syntax.
- A set of solution algorithms that exploit the syntactic structure.

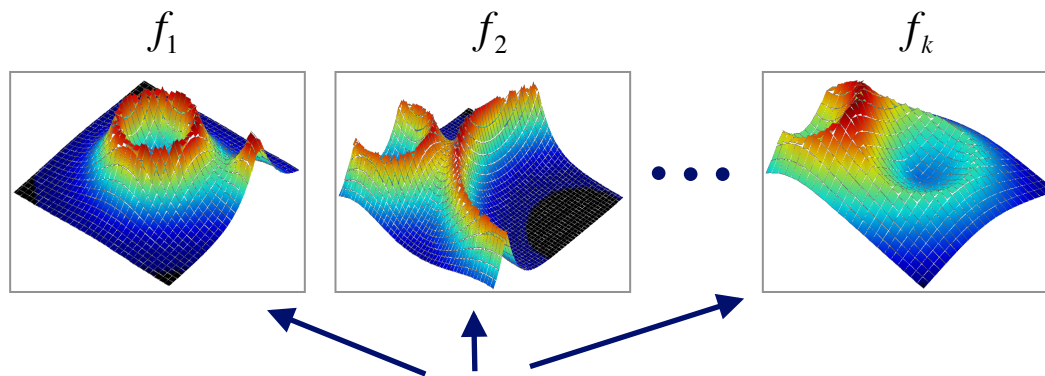


The “IvP Helm” is a MOOS App implementing a behavior-based architecture for autonomous decision making, using Interval Programming to reconcile competing behaviors.

# Interval Programming

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

- The solution,  $\vec{x}^*$ , is the single decision that maximizes the weighted sum of all utility functions.
- For example, the best combination of vehicle heading, speed, and depth.

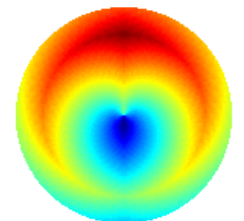
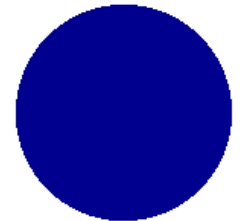


- Each function is the output of a *behavior*.
- Each function is a utility function – a mapping from possible decisions to their relative merit.
- Freedom from function form assumptions is key to solving general autonomy problems.
- Global optimality guaranteed.

• New problem generated and solution found on each iteration of the decision loop, typically 4x per second.

Obstacle Vehicle

Waypoint

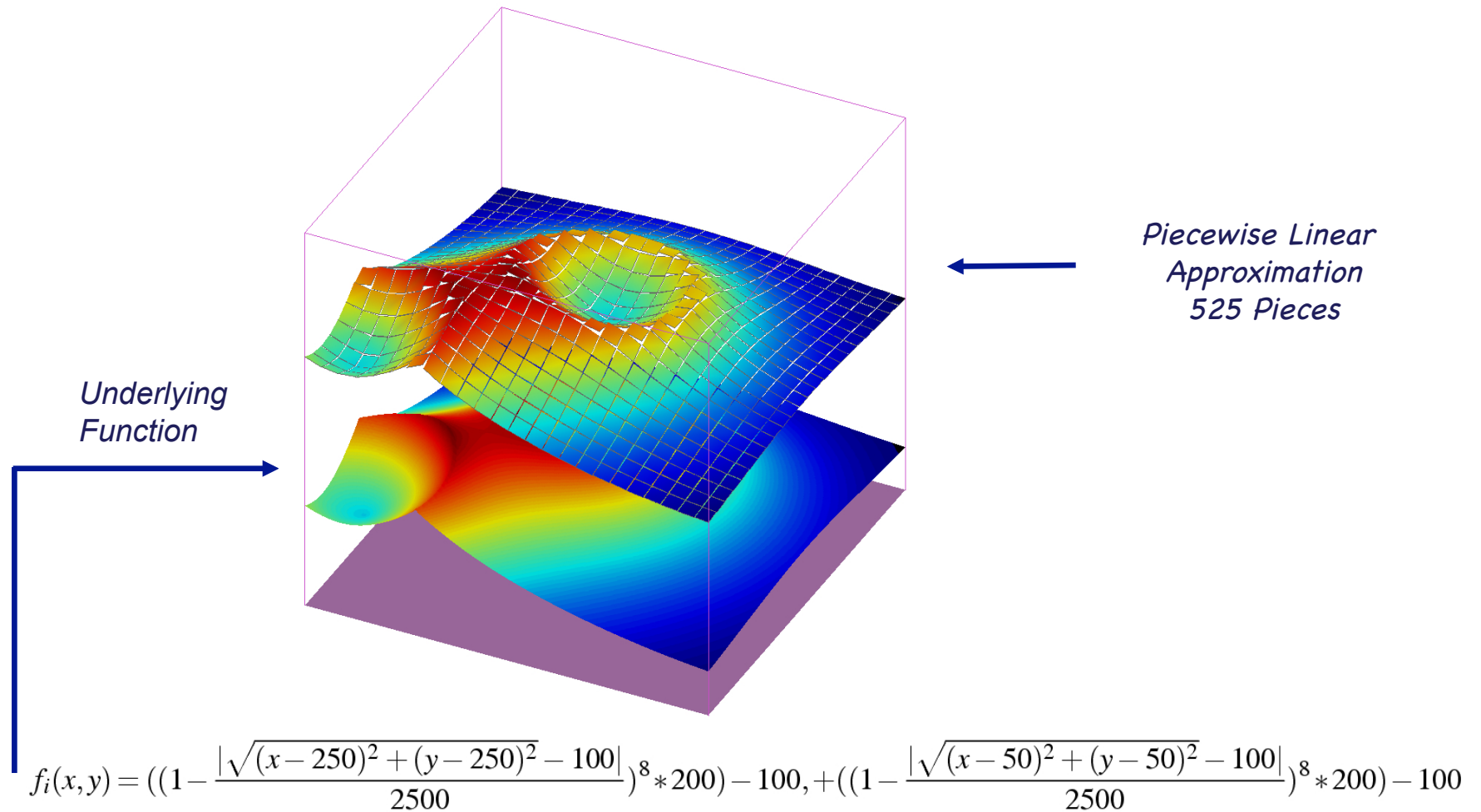


Controlled Vehicle



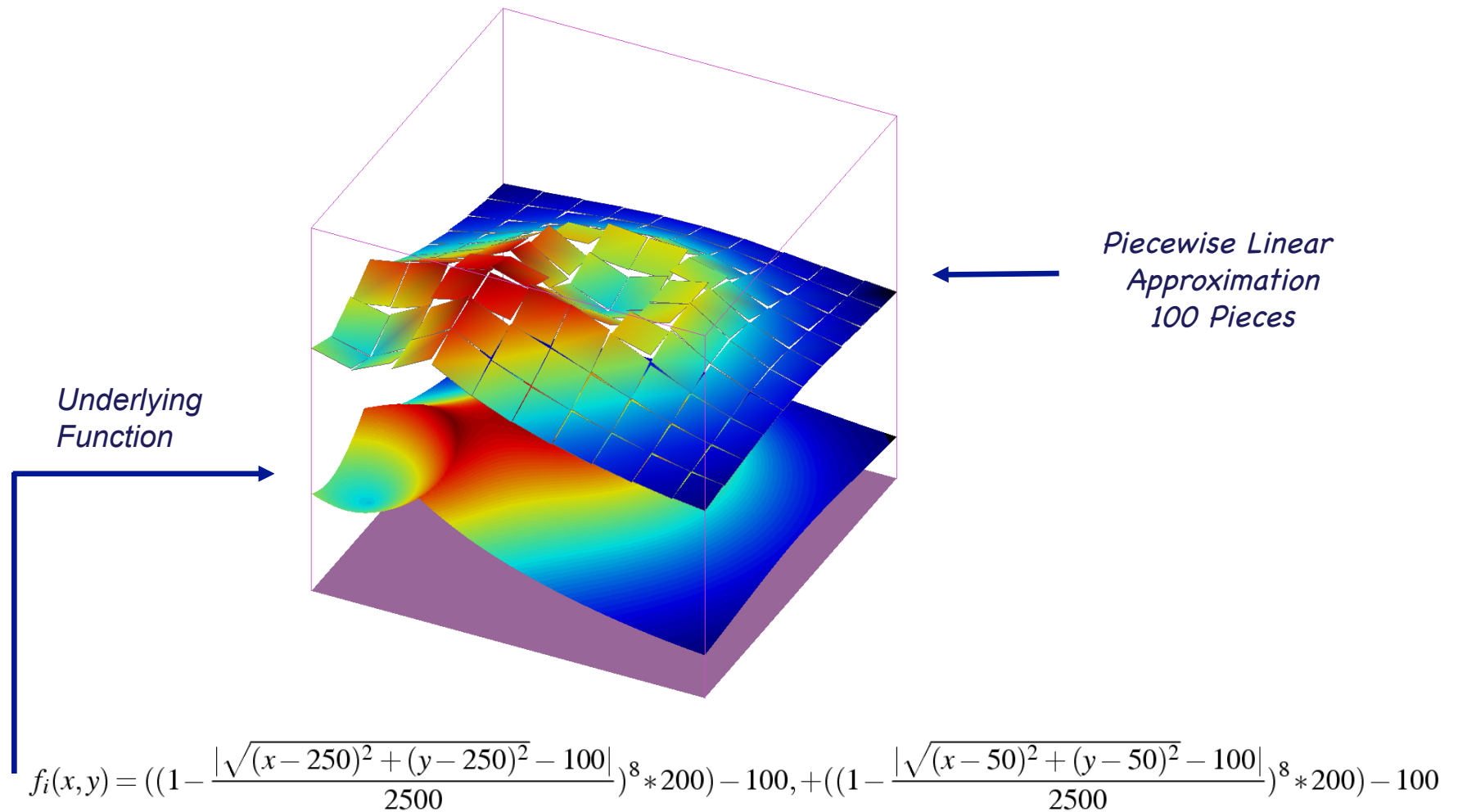
# IvP Functions

An **IvP Function** is a piecewise linear approximation of an objective function, over a discrete decision space.



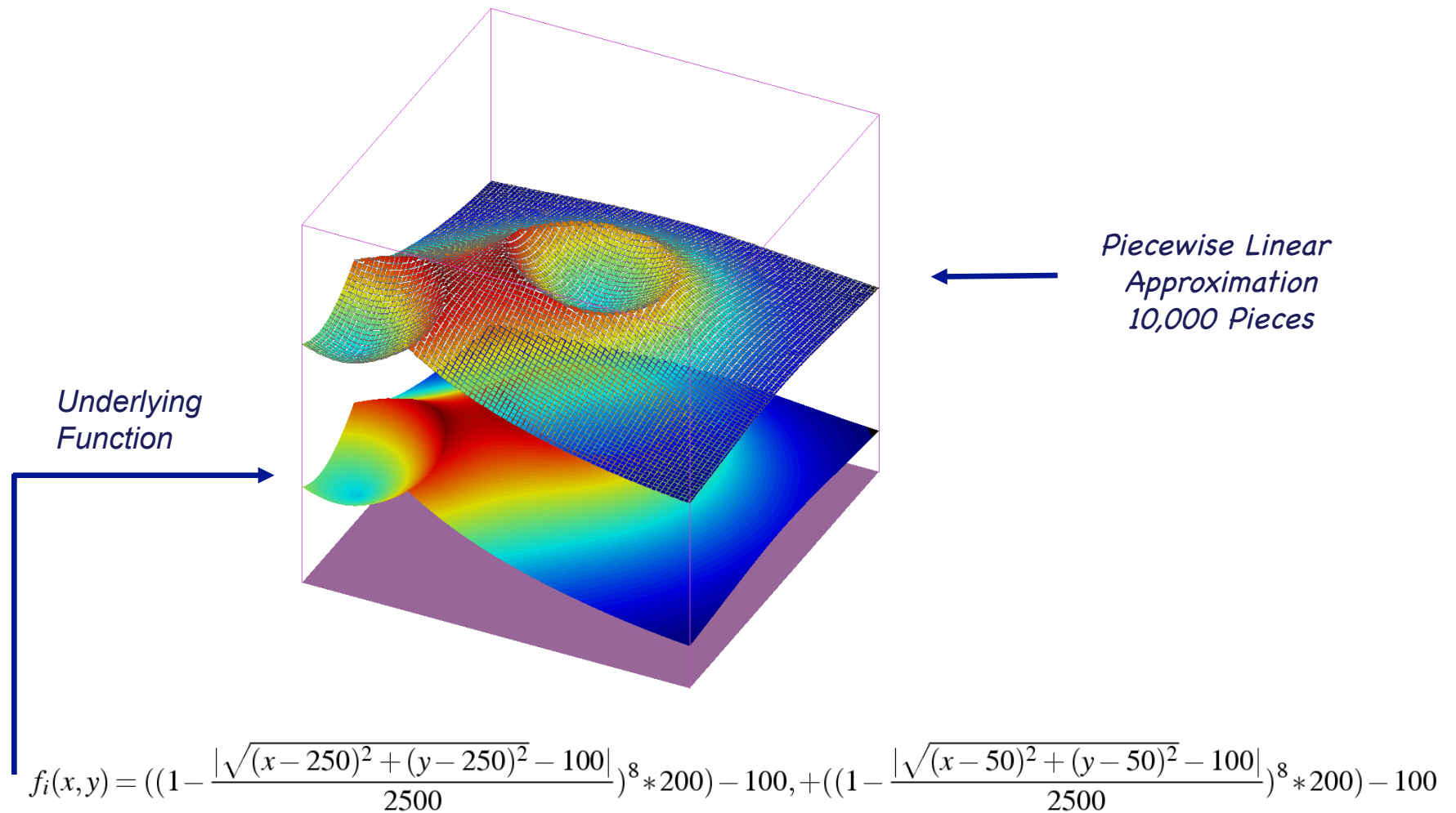
# IvP Functions

An **IvP Function** is a piecewise linear approximation of an objective function, over a discrete decision space.



# IvP Functions

An **IvP Function** is a piecewise linear approximation of an objective function, over a discrete decision space.



## Piecewise linear (IvP) functions:

- Each point in the decision space belongs to exactly one piece.
- Each pieces has an interval boundary and a linear interior function.

## Advantages:

- Any underlying function can be represented.
- Pieces need not be uniformly distributed.
- Extends to  $n$  dimensions.
- Syntax can be exploited by the solution algorithms.

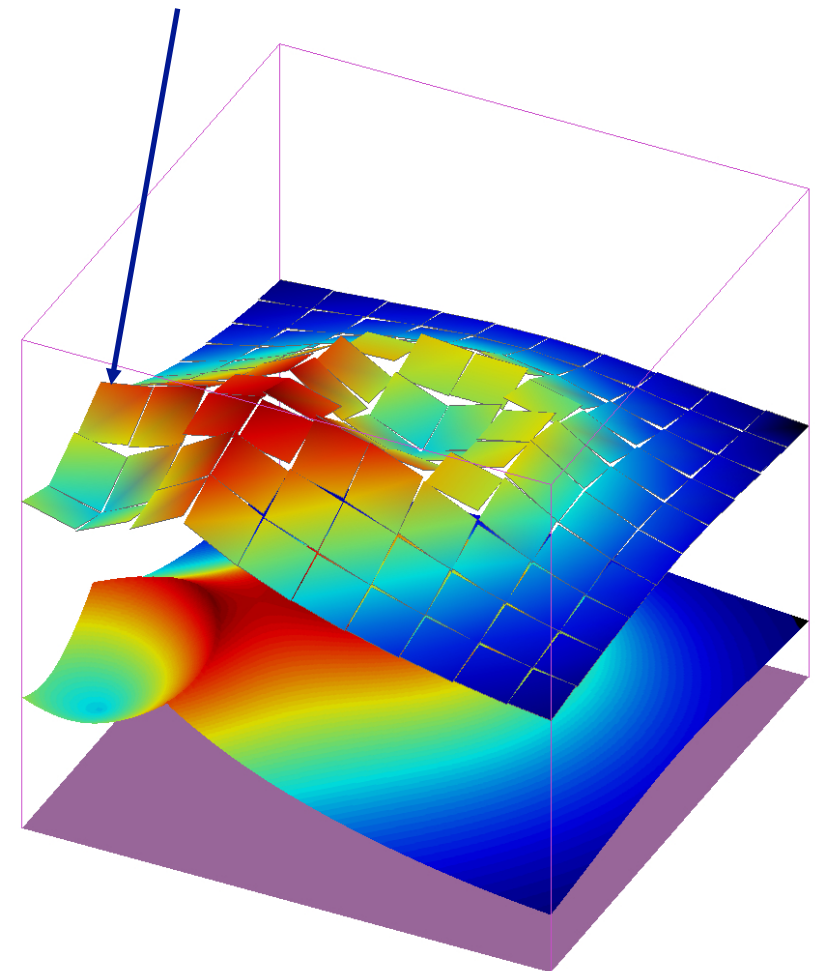
Interval Boundary:

$$10 \leq x \leq 20$$

$$14 \leq y \leq 21$$

Interior Function:

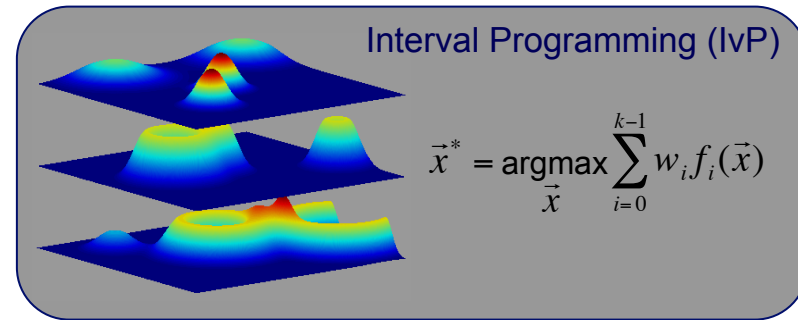
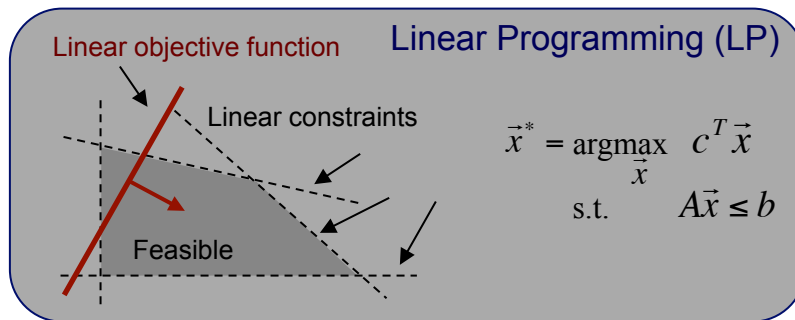
$$f(x,y) = 4x + 8y + 7$$



# Interval Programming

Interval Programming is a mathematical programming model.

- A mathematical structure or syntax for representing an optimization problem.
- • A toolbox of practical methods for casting different problems into the IvP syntax.
- A set of solution algorithms that exploit the syntactic structure.



## The IvPBuild Toolbox:

- A C++ library of tools for building syntactically correct IvP functions.
- Typically invoked from code within an IvP Helm behavior implementation
- Downloadable from [www.moos-ivp.org](http://www.moos-ivp.org)
- Documentation: <http://dspace.mit.edu/handle/1721.1/46361>



# The IvPBuild Toolbox



Q: How are IvP Functions built?

A: The [IvPBuild Toolbox](#)

The [IvPBuild Toolbox](#) is a

- C++ Library,
- Distributed with the MOOS-IvP tree.
- A set of tools for building IvP functions from a user's underlying objective function.
- Meant to be invoked from within a behavior implementation – from within the [onRunState\(\)](#) function.
  
- The [IvPBuild Toolbox](#) contains two basic tools:
- The [ZAIC](#) tool – for building 1D objective functions.
- The [Reflector](#) tool – for building IvP Functions in N dimensions.

# The Reflector Tool

## (Pure Uniform)

The *Reflector Tool* builds an IvP function from a given underlying function by sampling the underlying function.

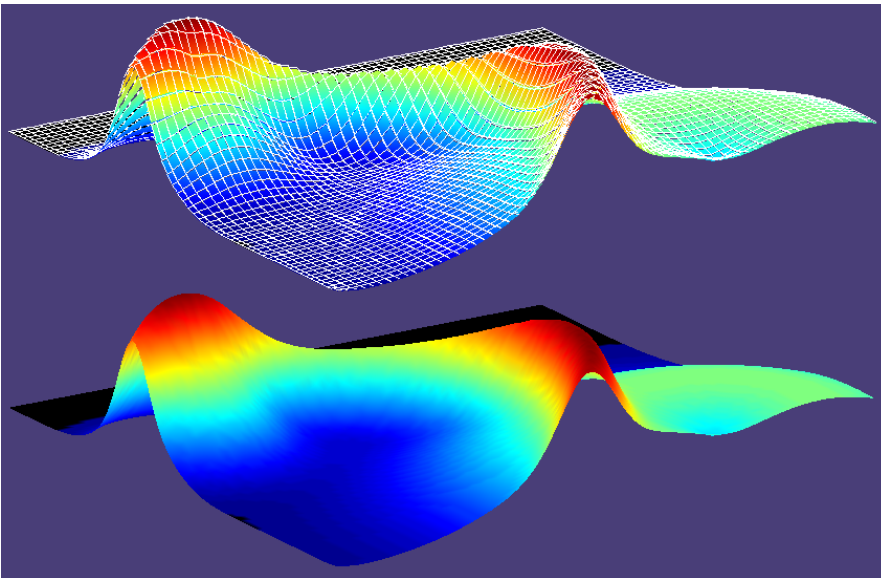
Three variations discussed:

- (1) Pure Uniform
- (2) Uniform with Prioritized Augmentation
- (3) Uniform with Focused Augmentation

### Algorithm Design Criteria

- (1) Minimize error.
- (2) Minimize pieces generated.
- (3) Minimize generation time.
- ★ (4) Minimize complexity of use for the user.
- ★ (5) Minimize restrictiveness of the tool.

Method 1 - Pure Uniform



### Basic idea:

- Function is composed of uniform piecewise linearly defined pieces.

### Pros:

- Simple to use.
- Requires no insight into underlying function.
- Can explore time, size, accuracy tradeoff space.

### Cons:

- Treats all areas of the underlying function equally.
- Does not capitalize on insight into underlying function.

# The Reflector Tool

## (Uniform with Prioritized Augmentation)

The *Reflector Tool* builds an IvP function from a given underlying function by sampling the underlying function.

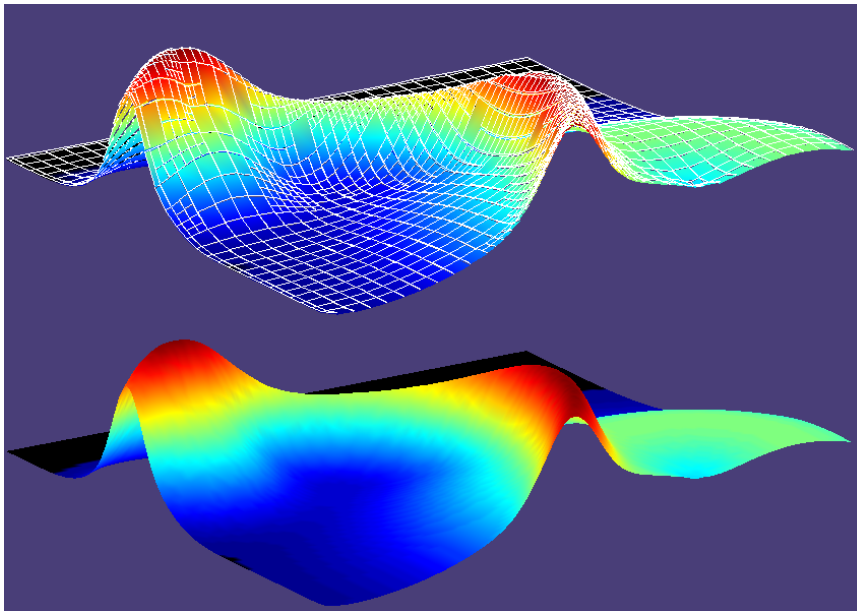
Three variations discussed:

- (1) Pure Uniform
- (2) Uniform with Prioritized Augmentation
- (3) Uniform with Focused Augmentation

### Algorithm Design Criteria

- ★ (1) Minimize error.
- ★ (2) Minimize pieces generated.
- (3) Minimize generation time.
- ★ (4) Minimize complexity of use for the user.
- (5) Minimize restrictiveness of the tool.

### Method 2 - Uniform with Priority-Based Augmentation



### Basic idea:

Start with a uniform function and further refine the pieces that have the worst error (prioritized during first linear regression phase).

### Pros:

- Simple to use. No insight into underlying function required
- Can explore time, size, accuracy tradeoff space.

### Cons:

- Does not always catch the pieces with worst error.
- Does not capitalize on insight into underlying function.



# The Reflector Tool

(Uniform with Focused Augmentation)

The *Reflector Tool* builds an IvP function from a given underlying function by sampling the underlying function.

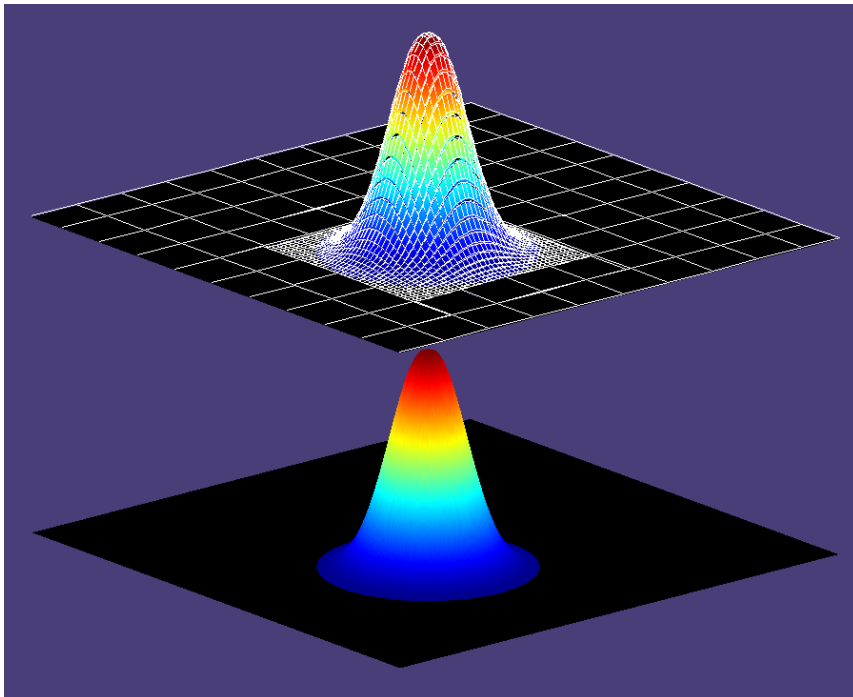
Three variations discussed:

- (1) Pure Uniform
- (2) Uniform with Prioritized Augmentation
- (3) Uniform with Focused Augmentation

## Algorithm Design Criteria

- ★ (1) Minimize error.
- ★ (2) Minimize pieces generated.
- ★ (3) Minimize generation time.
- (4) Minimize complexity of use for the user.
- (5) Minimize restrictiveness of the tool.

### Method 3 - Uniform with Focused Augmentation



### Basic idea:

Start with a uniform function and further refine the pieces in areas thought to need more pieces for error reduction

### Pros:

- Simple to use. Capitalizes on insight of underlying function.
- Can explore time, size, accuracy tradeoff space.

### Cons:

- Not all functions have area suitable for focused refinement.
- Requires insight into underlying function.

# The ZAIC Tool

The *ZAIC Tool* builds an IvP function from a given underlying function by sampling the underlying function.

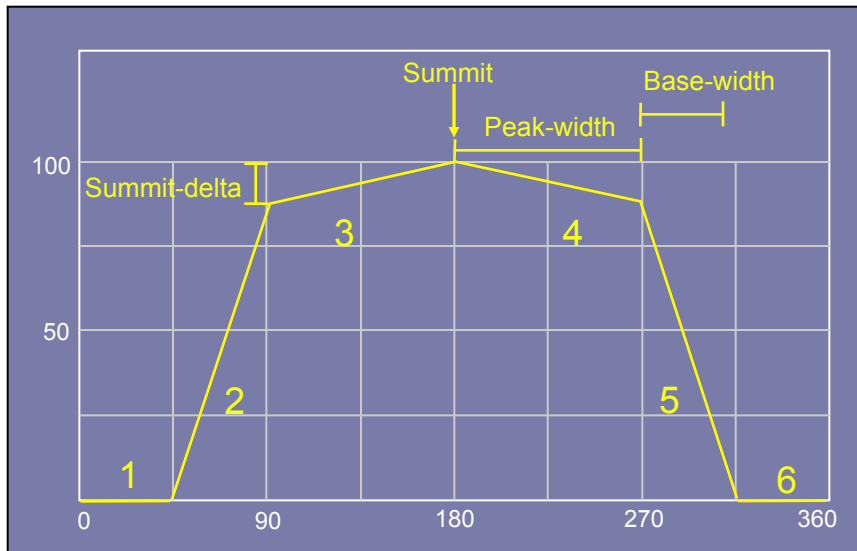
Three variations:

- (1) ZAIC\_PEAKEs
- (2) ZAIC\_LEQ
- (3) ZAIC\_HEQ

## Algorithm Design Criteria

- ★ (1) Minimize error.
- ★ (2) Minimize pieces generated.
- ★ (3) Minimize generation time.
- ★ (4) Minimize complexity of use for the user.
- ★ (5) Minimize restrictiveness of the tool.

### Method 4 - ZAIC Peaks



Basic idea:

1D Functions with one or more peaks. Identify the peak properties and the IvP function is generated.

Pros:

Simple to use. Very few pieces.  
As many peaks as desired.

Cons:

Only suitable for 1D objective functions.

IvP Function:

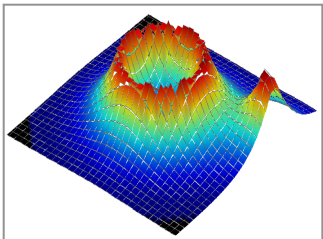
- |     |                       |                    |
|-----|-----------------------|--------------------|
| (1) | $0 \leq x \leq 45$    | $y = 0$            |
| (2) | $46 \leq x \leq 90$   | $y = 1.89x - 85$   |
| (3) | $91 \leq x \leq 180$  | $y = 0.17x + 70$   |
| (4) | $181 \leq x \leq 270$ | $y = -0.17x + 130$ |
| (5) | $271 \leq x \leq 315$ | $y = -1.89x + 595$ |
| (6) | $316 \leq x \leq 359$ | $y = 0$            |

# Interval Programming Solution Algorithms (Overview)

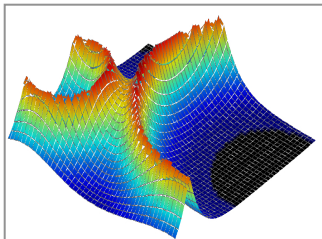
An *IvP* problem consists of a set of  $k$  functions, each with a priority weighting.  
The solution is given by:

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

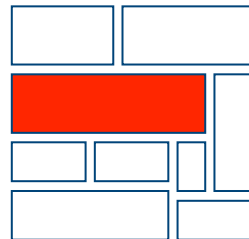
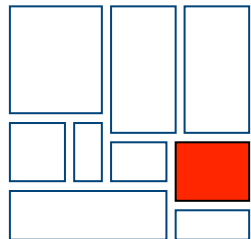
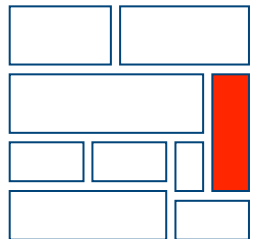
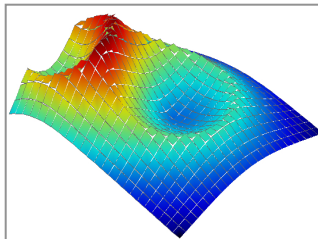
$f_1$



$f_2$



$f_3$

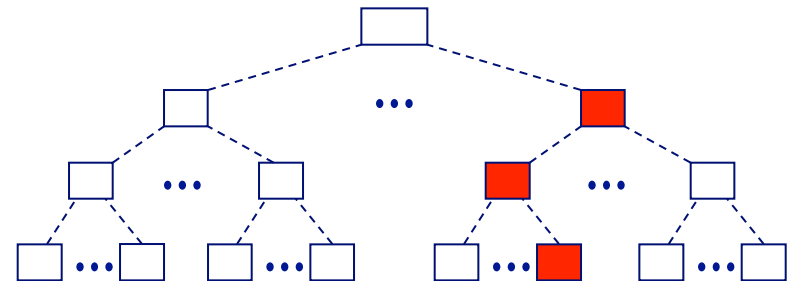


## The Search Tree:

- 1 level for each function
- $n^k$  leaf nodes ( $n$  pieces per function).

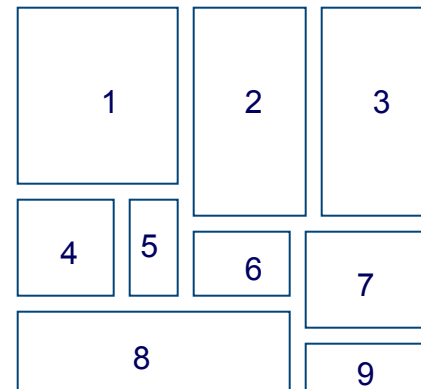
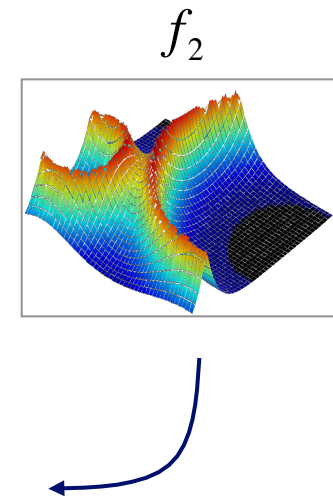
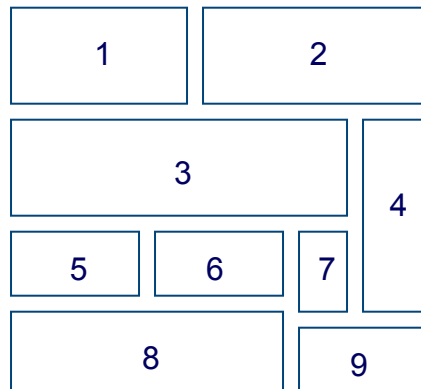
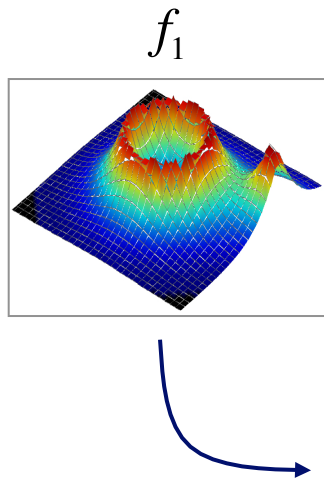
## The Solution algorithm:

- Branch and bound
- Pruning based on intersection look-ahead.



# IvP Solution Algorithms

(The issue of global optimality)



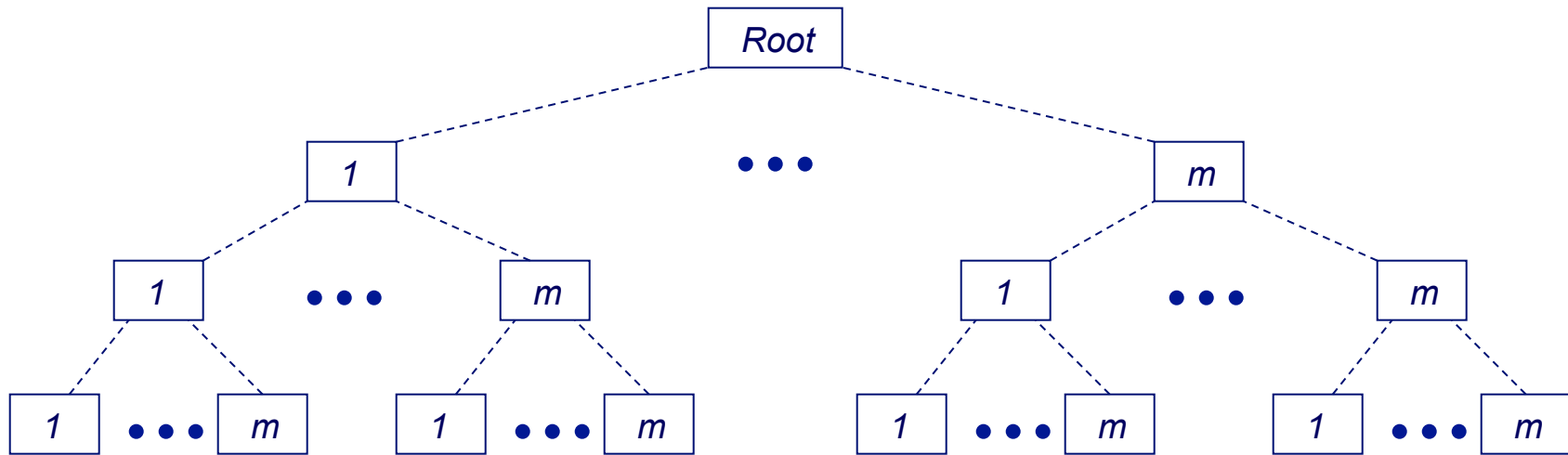
How global optimality is guaranteed:

- Each objective function is defined over the same decision space  $(x, y)$ , (heading, speed) etc.
- The solution space is the set of all possible combinations of pieces from each function.

e.g.,  $\{ (1,1), (1,2), (1, 3), (1,4), (1,5) \dots (9,5), (9,6), (9,7), (9,8), (9,9) \}$  81 total pairs

- Each point in the decision space belongs to exactly one piece from each piecewise function.
- Therefore considering all possible combinations ensures all possible decisions are considered.

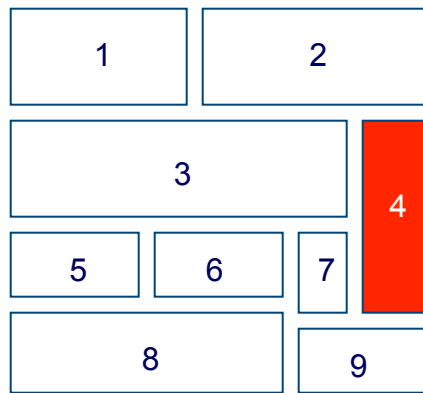
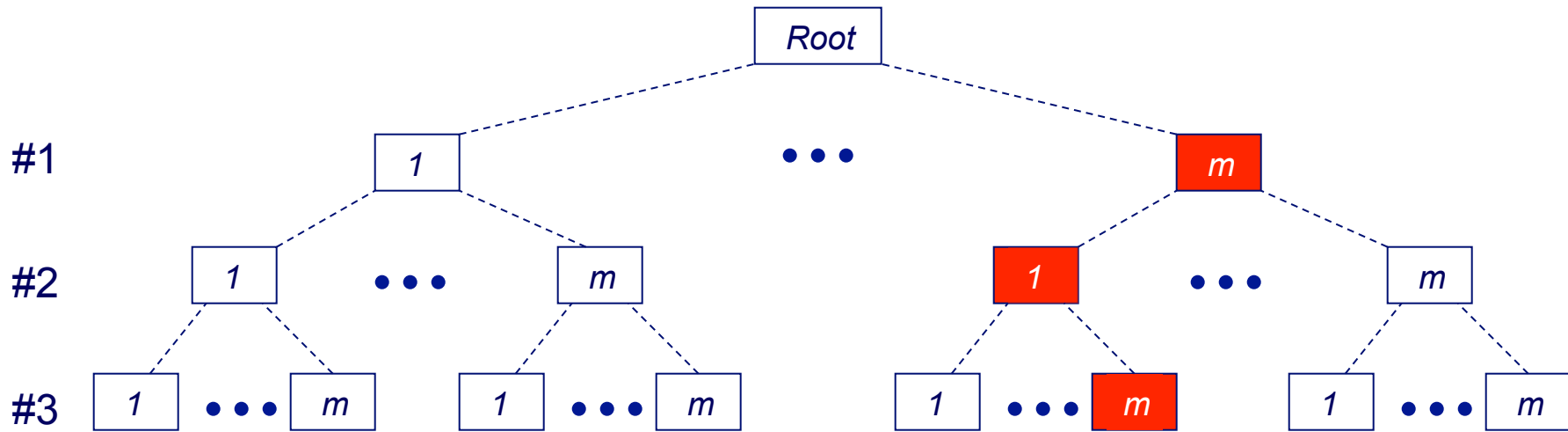
# IvP Solution Algorithms (The Search Tree)



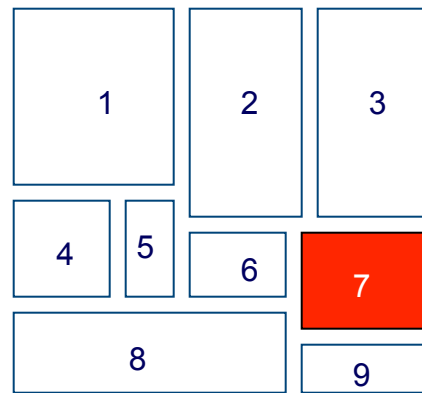
- Each level in the tree corresponds to one objective function.
- Each leaf node is a combination of one piece from each objective function.
- Function weights applied a priori to the pieces - weights no longer relevant.

$k$  functions,  $m$  pieces:  $m^k$  leaf nodes!

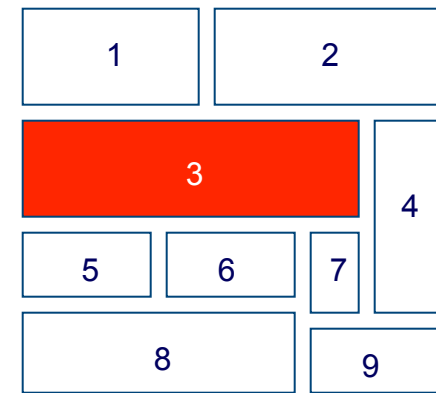
# Closer Look at a Leaf Node



#1



#2



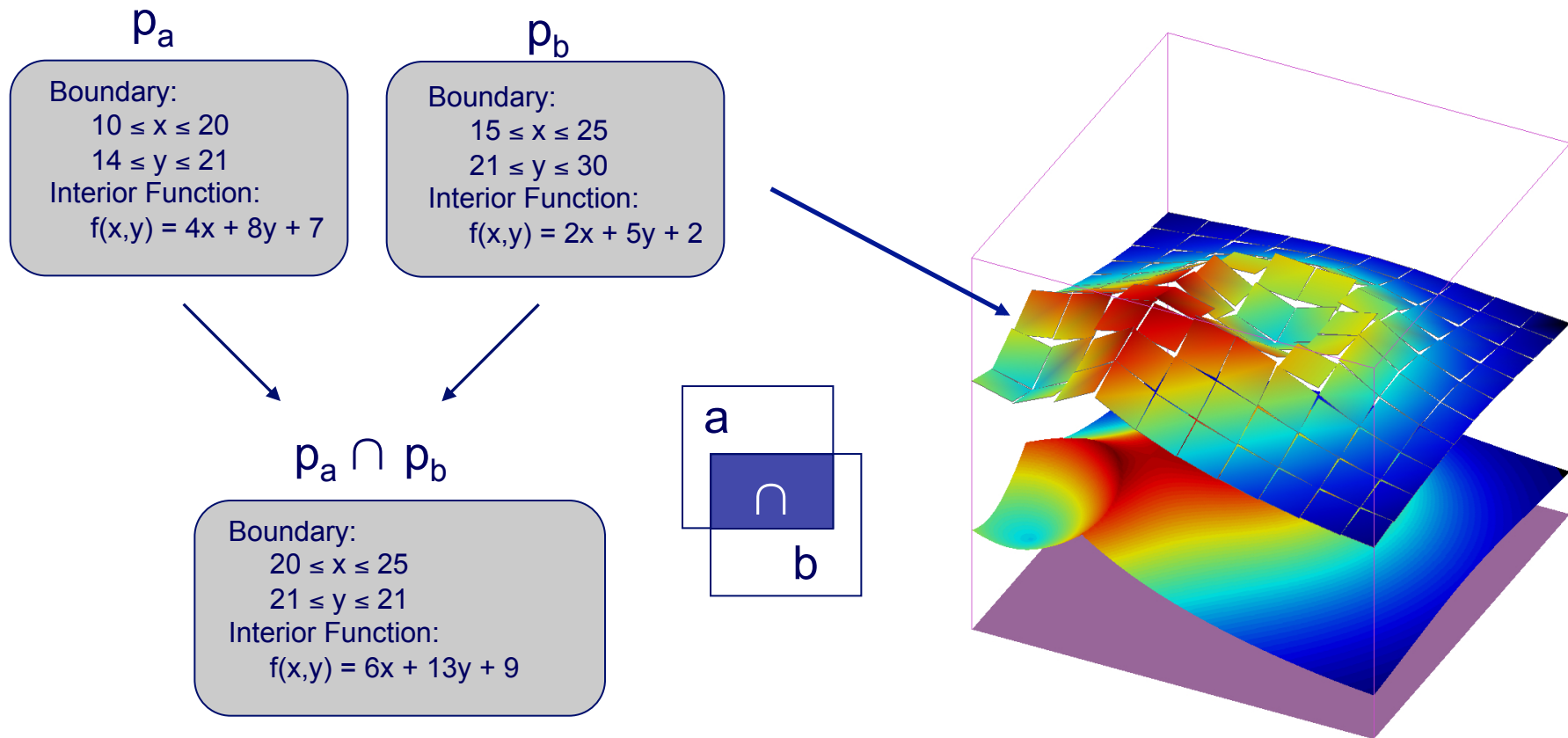
#3

# IvP Solution Algorithms

## (Piece Intersection)

Two pieces intersect if their boundaries overlap.

The result is a new piece with common boundary and the sum of their two linear interior functions.



$p_a \cap p_b$

Boundary:  
 $20 \leq x \leq 25$   
 $21 \leq y \leq 21$   
 Interior Function:  
 $f(x,y) = 6x + 13y + 9$

# IvP Solution Algorithms

(Full Search Tree Expansion)

```

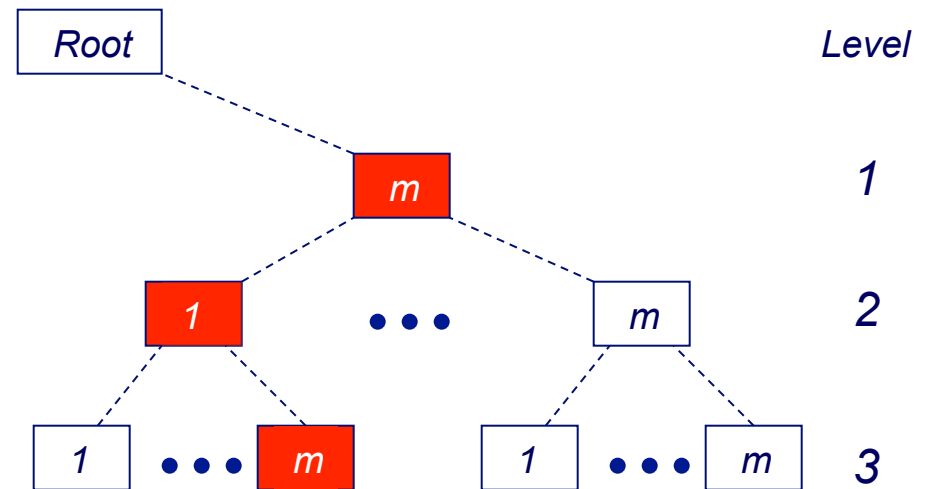
main()
0. bestBox = NULL
1. nodeBox = universe
2. search(nodeBox, 1)
3. return(bestBox)
    
```

**Recursive entry**

```

search(nodeBox, level)
0. if(level == k)
1.     if(nodeBox is non-empty)
2.         if(!bestBox or nodeBox→maxval > bestVal)
3.             bestVal = nodeBox→maxval
4.             bestBox = nodeBox
5.     return
6. for(i=1 to m)
7.     newNodeBox = nodeBox ∩ p(i, level)
8.     search(newNodeBox, level + 1)
    
```

**Recursive call**



**We can do better here**

**Don't recurse if newNodeBox is empty**



# IvP Solution Algorithms (Simple Pruning)

```

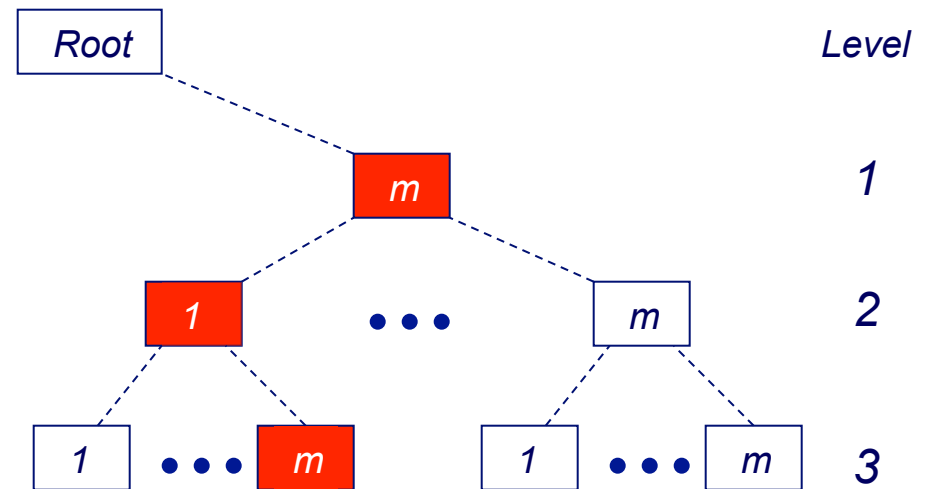
main()
0. bestBox = NULL
1. nodeBox = universe
2. search(nodeBox, 1)
3. return(bestBox)
  
```

**Recursive entry**

```

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

**Recursive call**

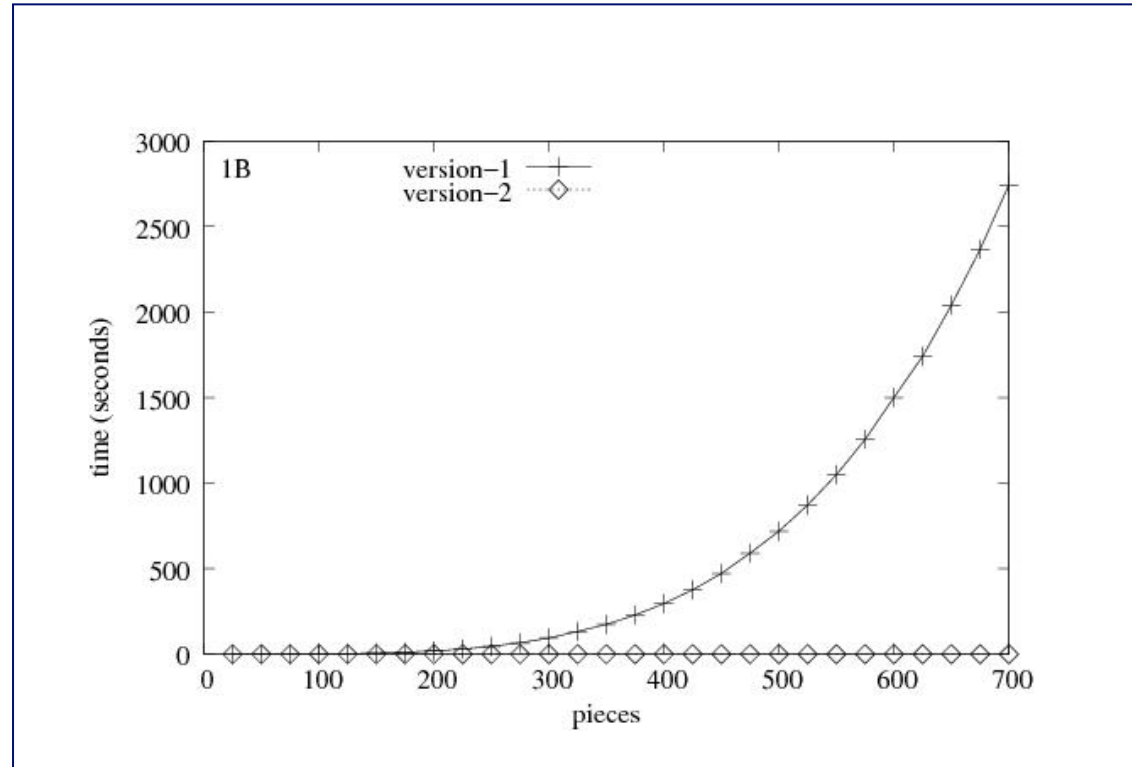


**Improved!**

**Don't recurse if newNodeBox is empty**

# IvP Solution Algorithms

(Full Expansion vs. Simple Pruning)



Each problem contains  
 3 objective functions  
 3 dimensions

# IvP Solution Algorithms

(Simple Pruning)

```

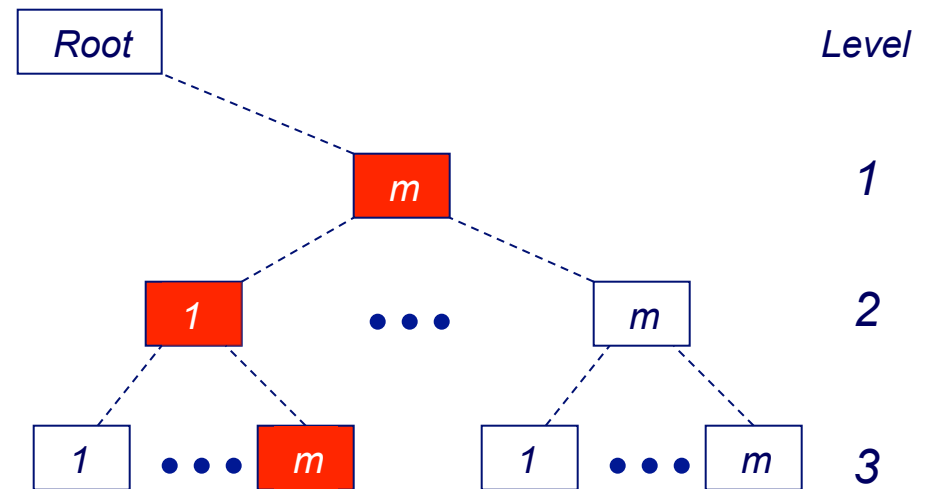
main()
0. bestBox = NULL
1. nodeBox = universe
2. search(nodeBox, 1)
3. return(bestBox)
  
```

**Recursive entry**

```

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

**Recursive call**



**Improve here:**

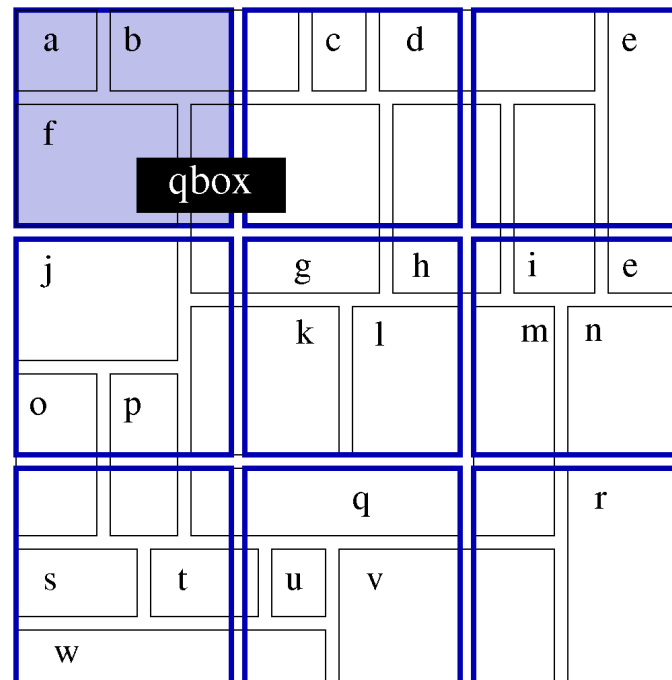
**Only intersect-test for pieces  
“in the neighborhood”.**

# IvP Solution Algorithms

(Efficient Intersection Detection)

Only intersect-test for pieces  
“in the neighborhood”.

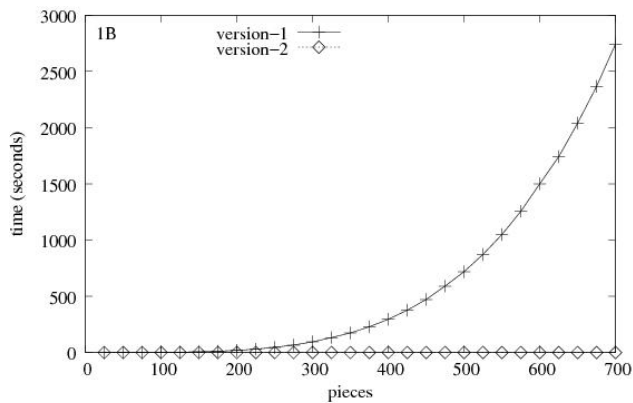
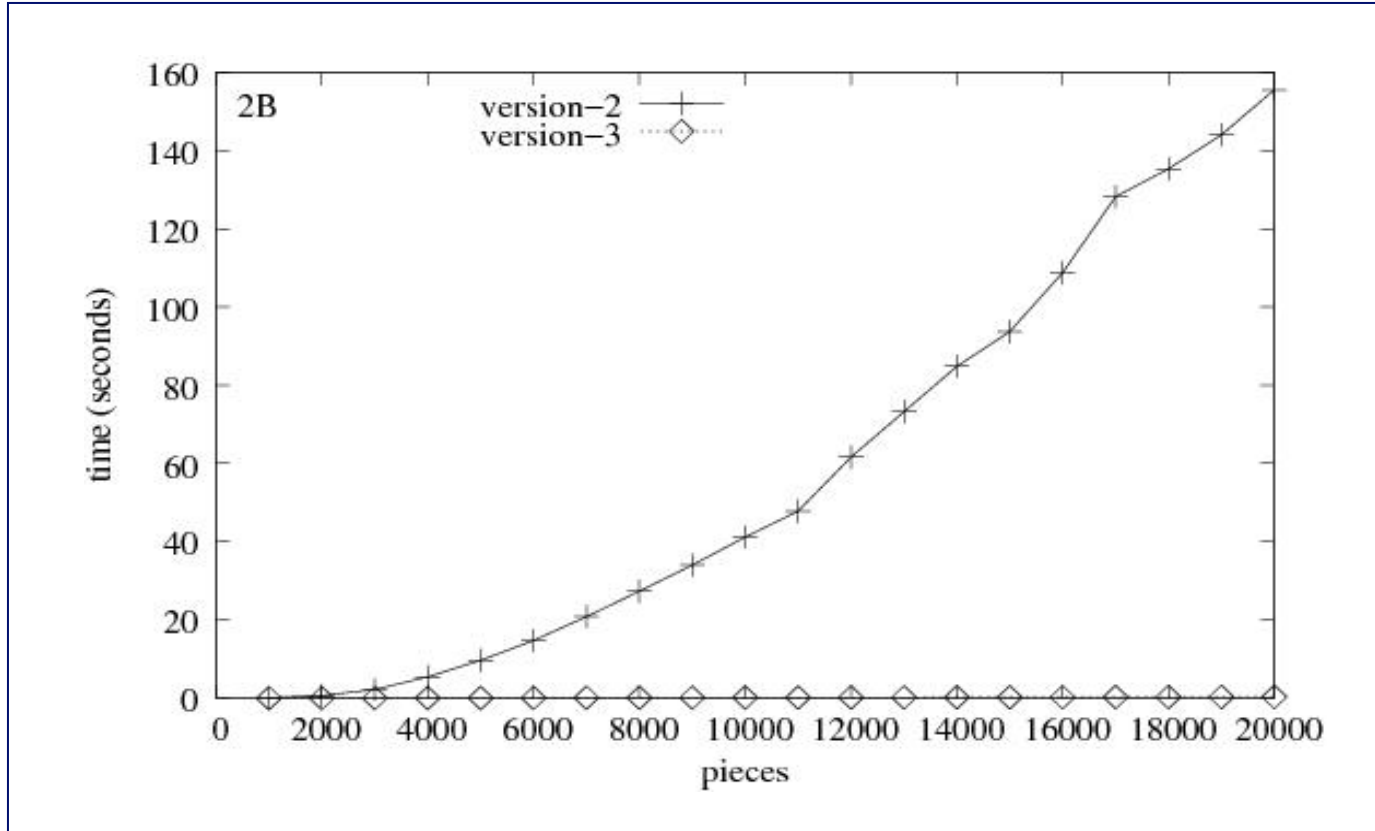
- A grid is associated with each piecewise objective function.
- A list is associated with each grid element containing pieces that intersect that element.
- The grid is populated when the function is constructed before the solution phase begins.
- The grid configuration can be different for each objective function.



Grid(level)→getBoxes(qbox) = {a, b, c, d, f, g, h}

# IvP Solution Algorithms

(Exhaustive vs. Grid-based Intersection Testing)

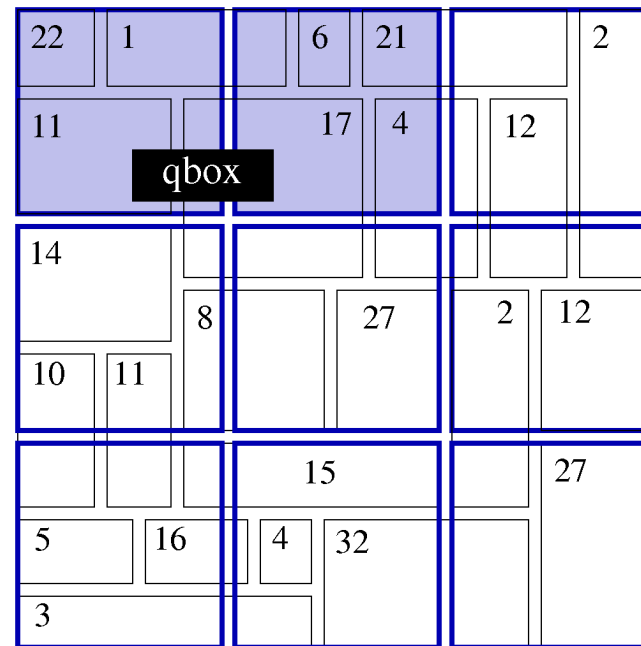


Each problem contains  
3 objective functions  
3 dimensions

## (Internal Node Upper Bounds)

### Get the upper bound for pieces “in the neighborhood”.

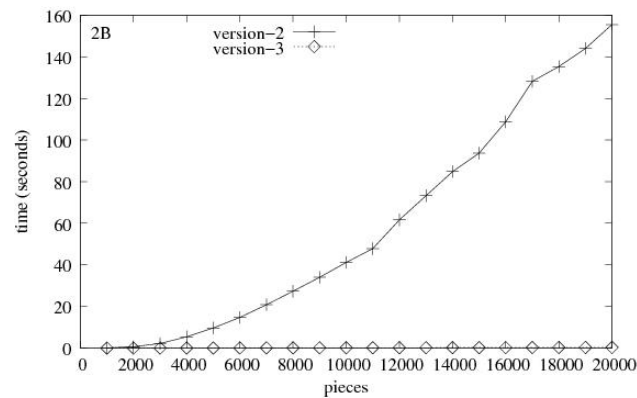
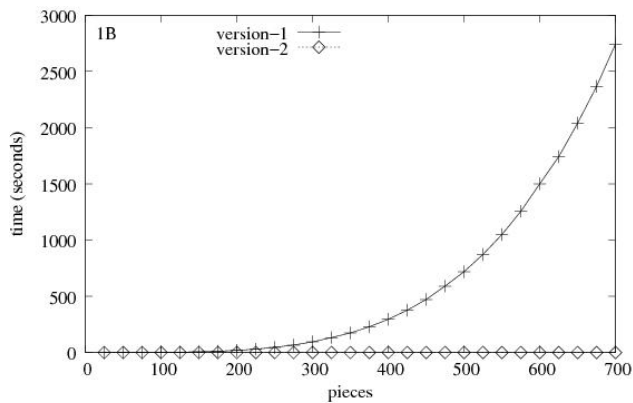
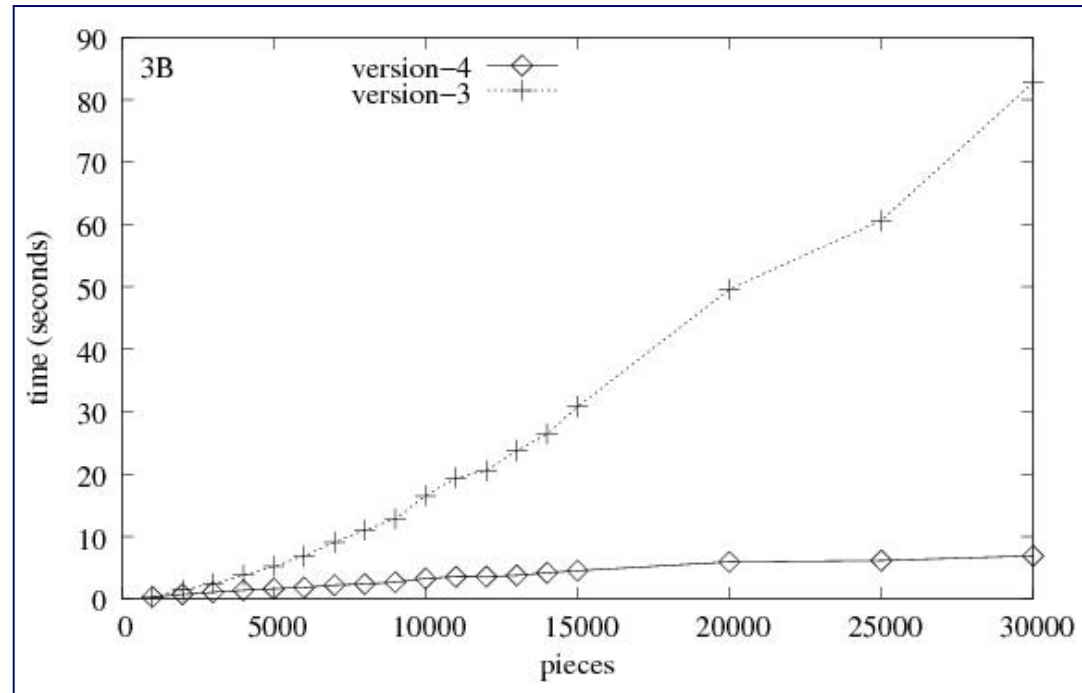
- Each piece has a known maximum value (of its linear interior function).
- Each grid element stores the max value of all pieces added to that element
- The grid is populated when the function is constructed before the solution phase begins.
- The grid configuration can be different for each objective function.



grid(level)→getBound(qbox) = 22

# IvP Solution Algorithms

(No-Upper-Bound vs. Upper-bound)



Each problem contains  
3 objective functions  
3 dimensions

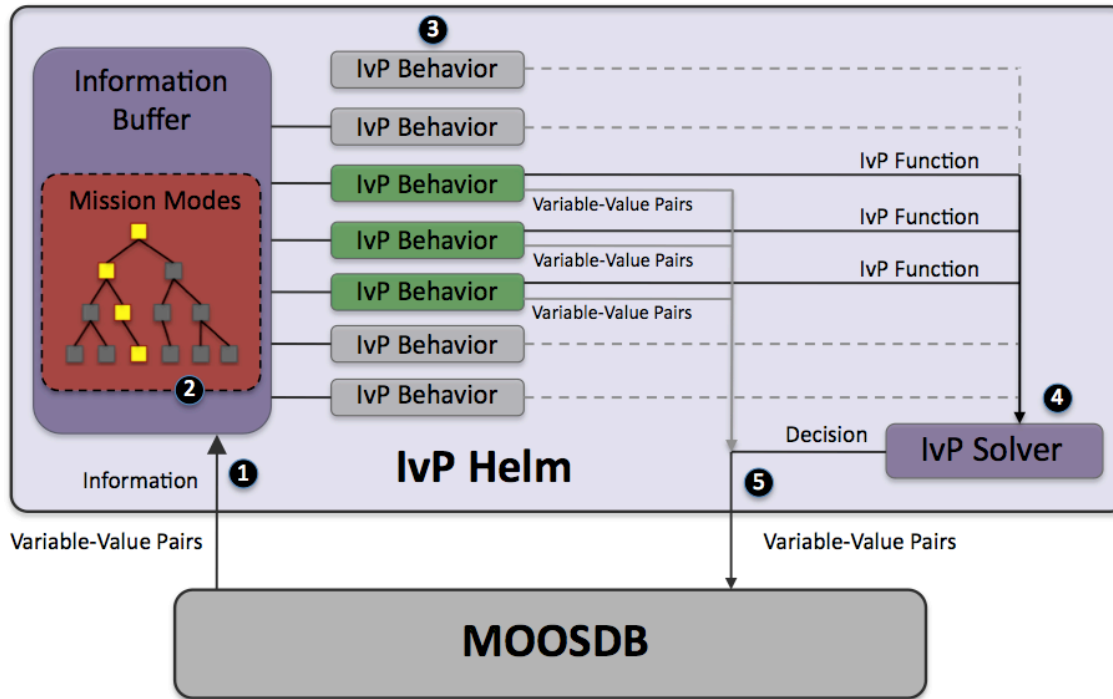


# Outline



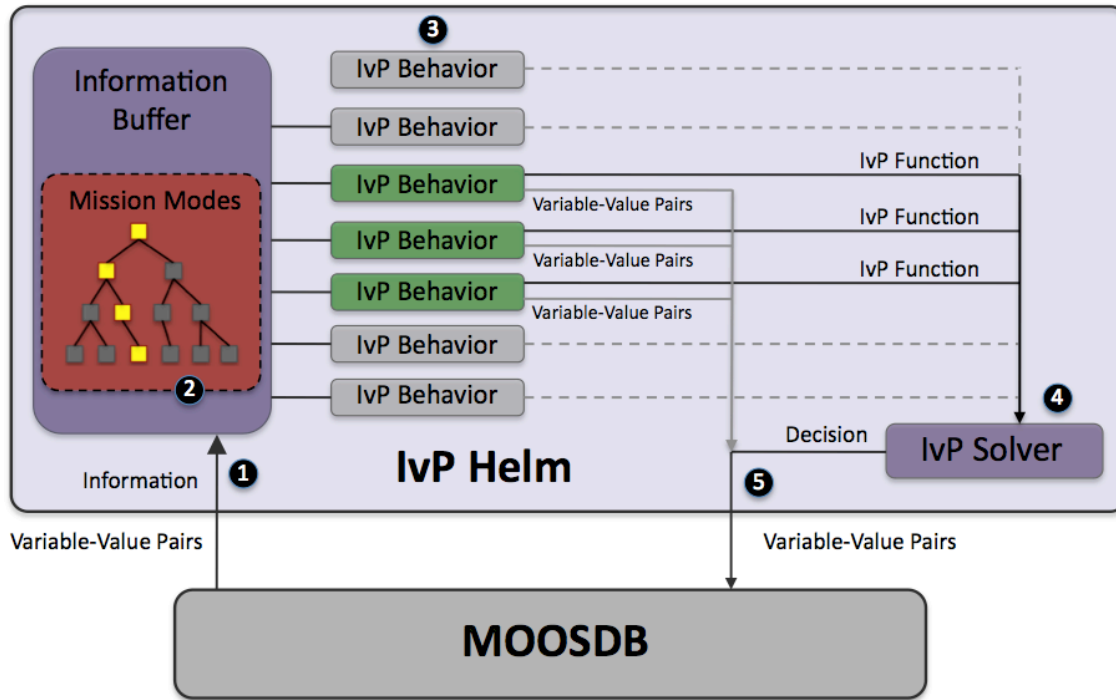
- Trends in autonomous marine vehicles
- The Payload Autonomy Paradigm and the MOOS-IvP project
- Multi-Objective Optimization with Interval Programming
- The IvP Helm
- MOOS-IvP 4.2 and Plans for Future Development





- 1 Mail is read in the MOOS OnNewMail() function and applied to a local buffer.
- 2 The helm mode is determined, and set of running behaviors determined.
- 3 Behaviors do their thing – posting MOOS variables and an IvP function.
- 4 Competing behaviors are resolved with the IvP solver.
- 5 The Helm decision and any behavior postings are published to the MOOSDB.

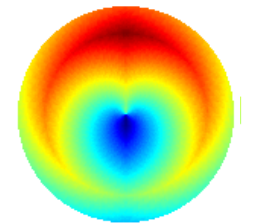
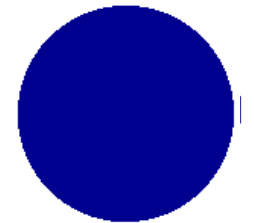
# Interval Programming and the IvP Helm



- 1 Mail is read.
- 2 Helm mode is determined.
- 3 Behaviors generate their output.
- 4 Competing behaviors are resolved.
- 5 The Helm posts its results

Obstacle Vehicle

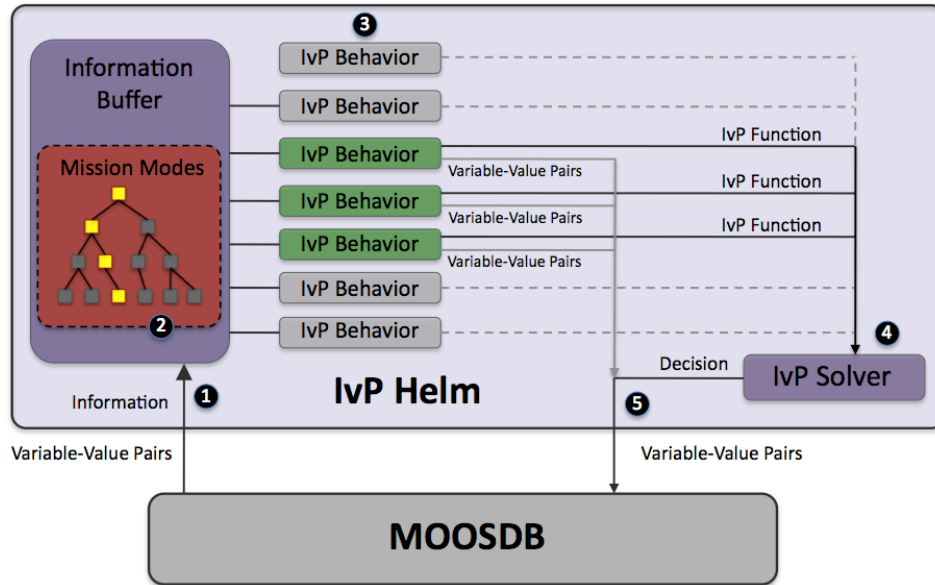
Waypoint



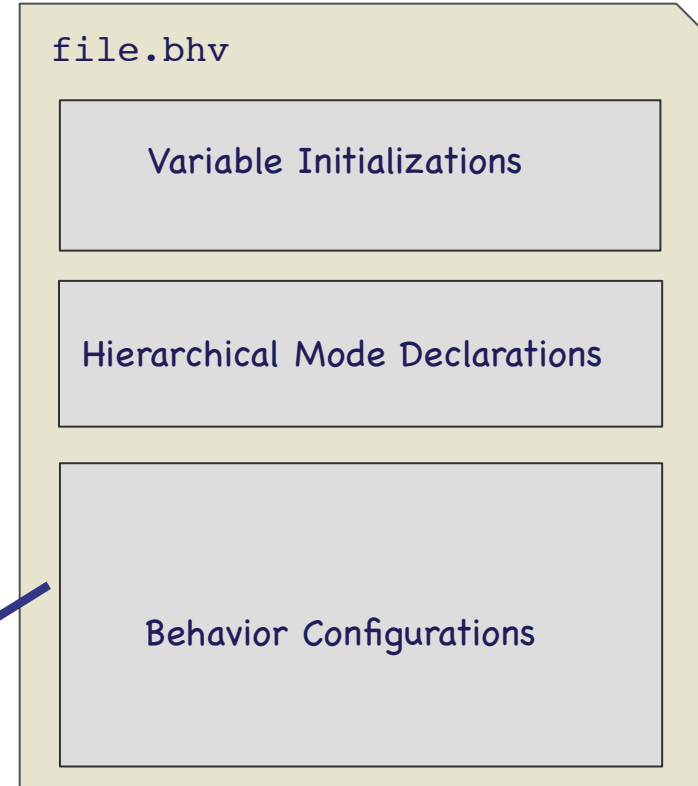
Controlled Vehicle



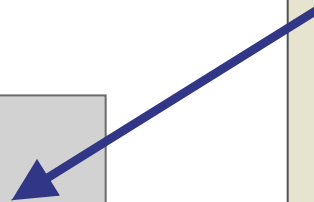
# IvP Helm Configuration

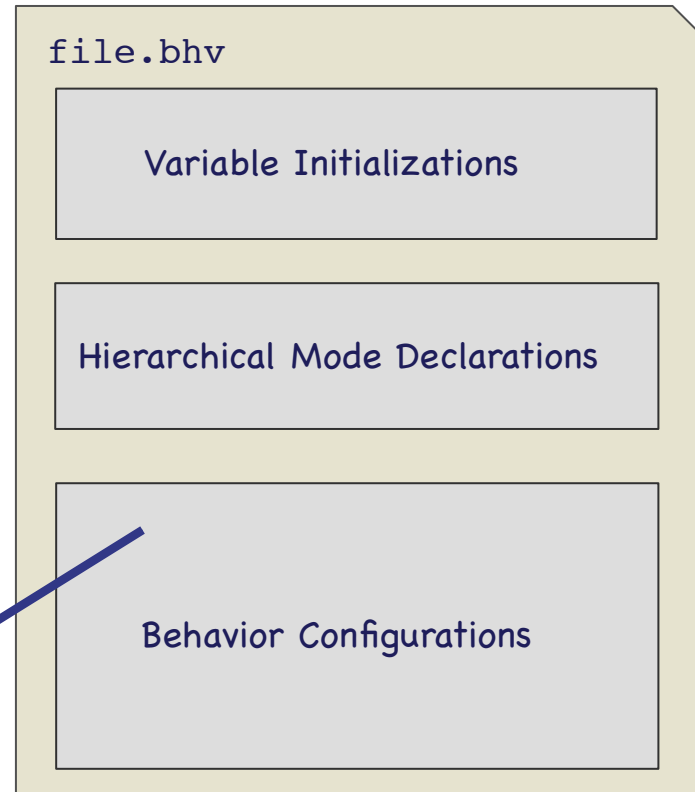
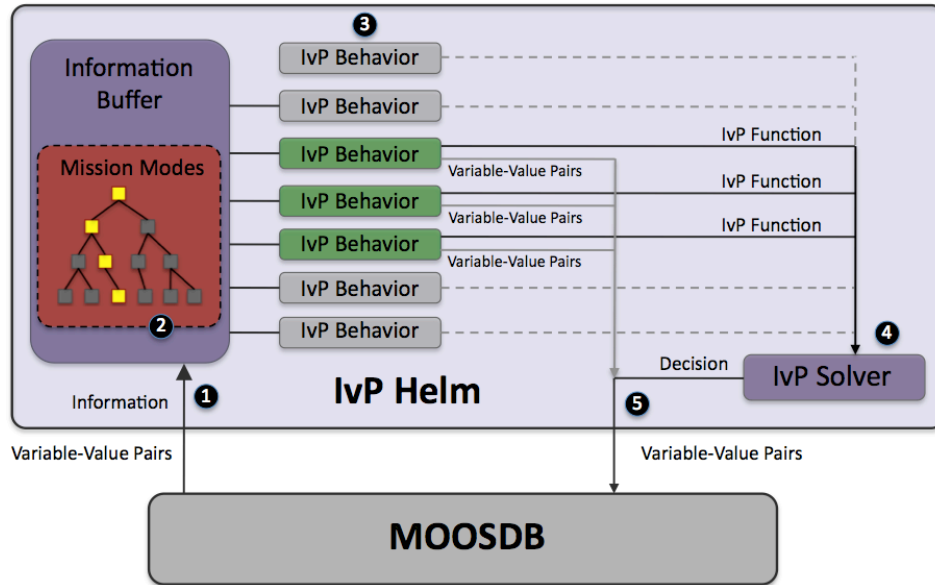


Helm configuration file structure:



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





```
Behavior = BHV_Loiter
{
  name      = loiter
  priority  = 100
  condition = (DEPLOY=true) and (REGION=A)

  speed = 1.8
  clockwise = false
  radius = 4.0
  nm_radius = 25.0
  polygon = format=radial, x=0, y=-75,
            radius=40, pts=8
}
```

# Simple Example: "Double Loiter"

## Mission Synopsis:

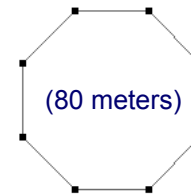
Upon receiving a deploy command, transit to and loiter at region A for a fixed duration and then to region B. Periodically switch between regions until recalled home.

```
Behavior = BHV_Loiter
{
  parameter = value
  . . .
  parameter = value
}
```

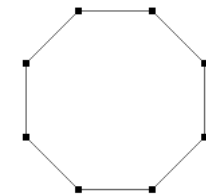
```
Behavior = BHV_Loiter
{
  parameter = value
  . . .
  parameter = value
}
```

```
Behavior = BHV_Return
{
  parameter = value
  . . .
  parameter = value
}
```

Launch and return position



REGION A



REGION B

# Simple Example: "Double Loiter"

## Mission Synopsis:

Upon receiving a deploy command, transit to and loiter at region A for a fixed duration and then to region B. Periodically switch between regions until recalled home.

```
Behavior = BHV_Loiter
{
  name      = loiter_a
  condition = (DEPLOY=true) and (REGION=A)

  speed = 1.8
  radius = 4.0
  polygon = format=radial,x=0,y=-75,radius=40,pts=8
}
```

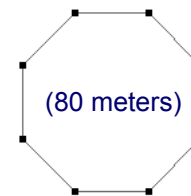
```
Behavior = BHV_Loiter
{
  name      = loiter_b
  condition = (DEPLOY=true) and (REGION=A)

  speed = 1.8
  radius = 4.0
  polygon = format=radial,x=160,y=-75,radius=40,pts=8
}
```

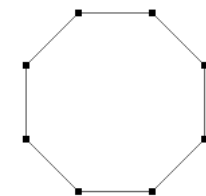
```
Behavior = BHV_Return
{
  name      = return
  condition = (DEPLOY=true) and (RETURN=true)

  speed = 1.8
  radius = 4.0
  point = 80,40
}
```

Launch and return position



REGION A



REGION B

# Simple Example: "Double Loiter"

## Mission Synopsis:

Upon receiving a deploy command, transit to and loiter at region A for a fixed duration and then to region B. Periodically switch between regions until recalled home.

```
Behavior = BHV_Loiter
{
  name      = loiter_a
  condition = (DEPLOY=true) and (REGION=A)

  speed = 1.8
  radius = 4.0
  polygon = format=radial,x=0,y=-75,radius=40,pts=8
}
```

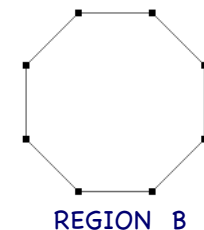
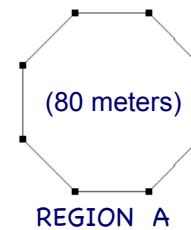
```
Behavior = BHV_Loiter
{
  name      = loiter_b
  condition = (DEPLOY=true) and (REGION=A)

  speed = 1.8
  radius = 4.0
  polygon = format=radial,x=160,y=-75,radius=40,pts=8
}
```

```
Behavior = BHV_Return
{
  name      = return
  condition = (DEPLOY=true) and (RETURN=true)

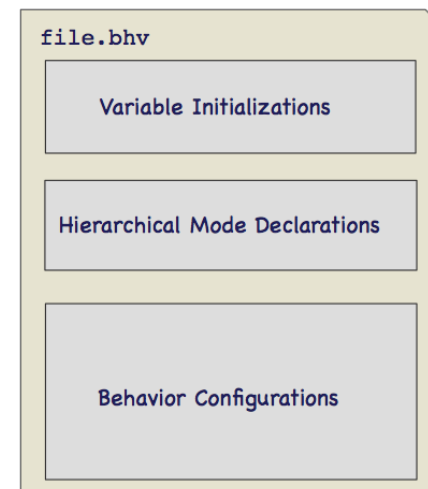
  speed = 1.8
  radius = 4.0
  point = 80,40
}
```

Launch and return position



```
Initialize DEPLOY = false
Initialize RETURN = false
Initialize REGION = A
```

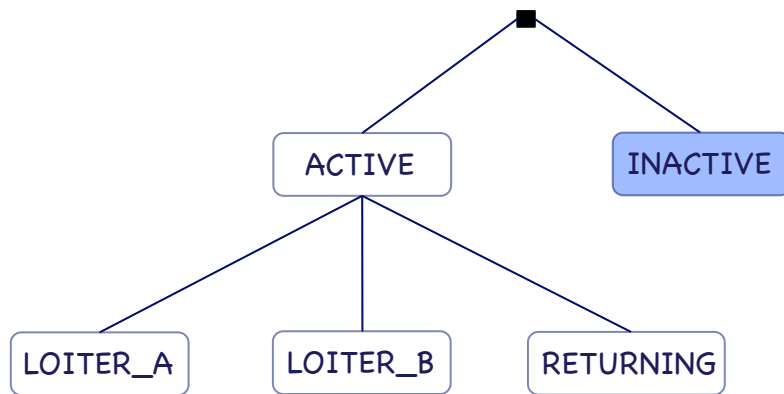
???



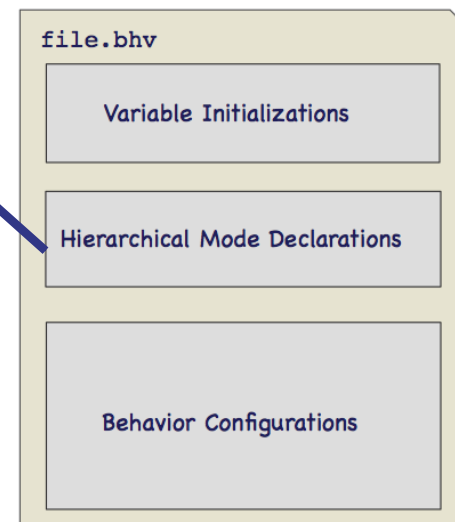
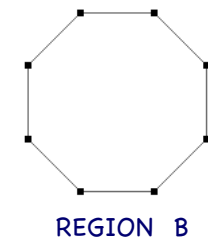
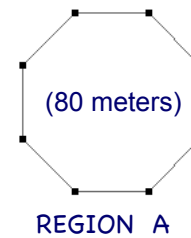
# Simple Example: "Double Loiter"

## Mission Synopsis:

Upon receiving a deploy command, transit to and loiter at region A for a fixed duration and then to region B. Periodically switch between regions until recalled home.



Launch and return position



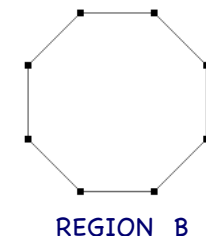
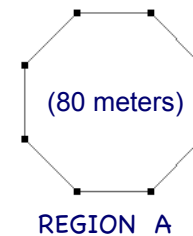
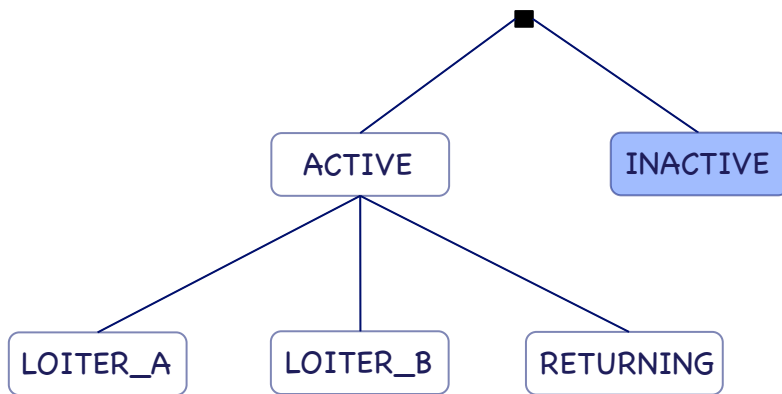


# Simple Example: "Double Loiter"

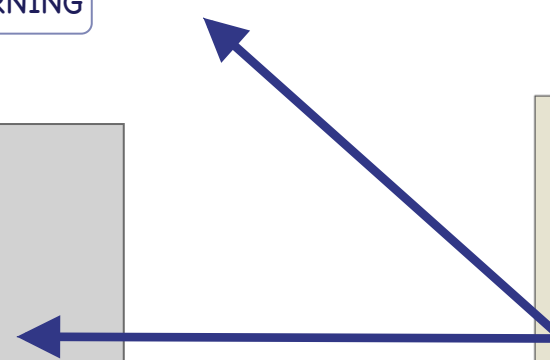
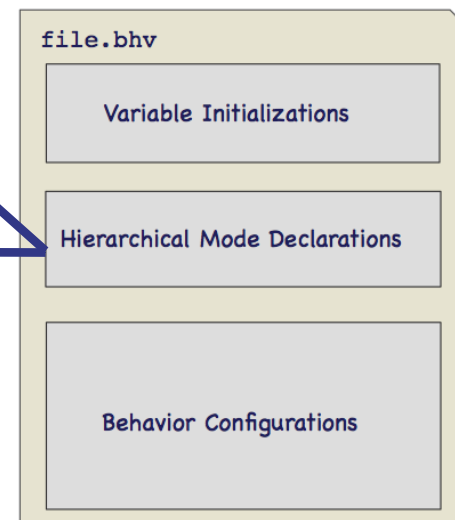
## Mission Synopsis:

Upon receiving a deploy command, transit to and loiter at region A for a fixed duration and then to region B. Periodically switch between regions until recalled home.

Launch and return position



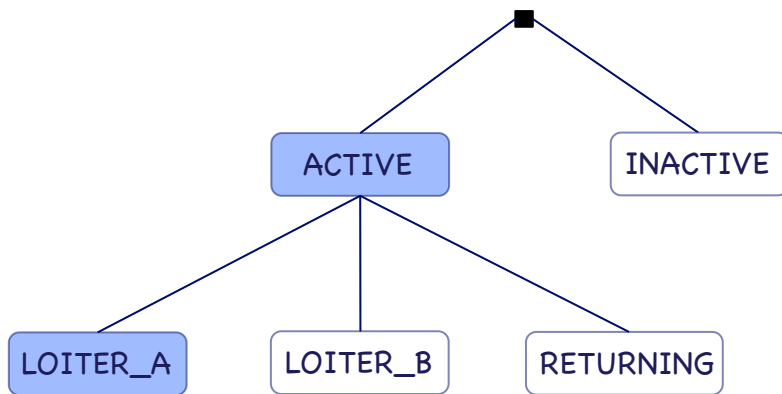
```
set MODE = ACTIVE {  
  DEPLOY = true  
} INACTIVE  
  
set MODE = RETURNING {  
  MODE = ACTIVE  
  RETURN = true  
}  
  
set MODE = LOITER_A {  
  MODE = ACTIVE  
  REGION = A  
} LOITER_A
```



# Simple Example: "Double Loiter"

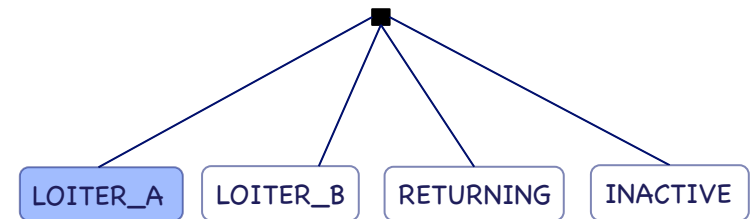
## Mission Synopsis:

Upon receiving a deploy command, transit to and loiter at region A for a fixed duration and then to region B. Periodically switch between regions until recalled home.



## Question:

Why define the "Active" mode?  
Why not just have:

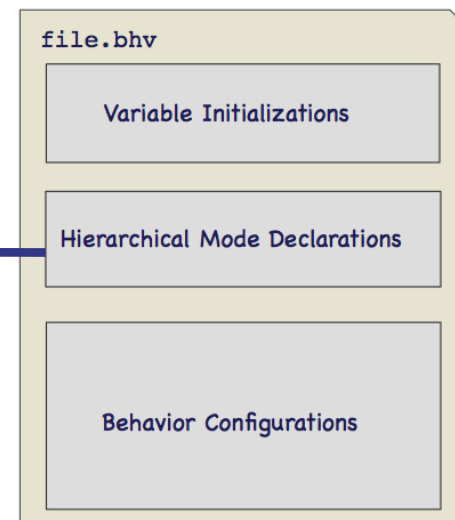


```

set MODE = ACTIVE {
  DEPLOY = true
} INACTIVE

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

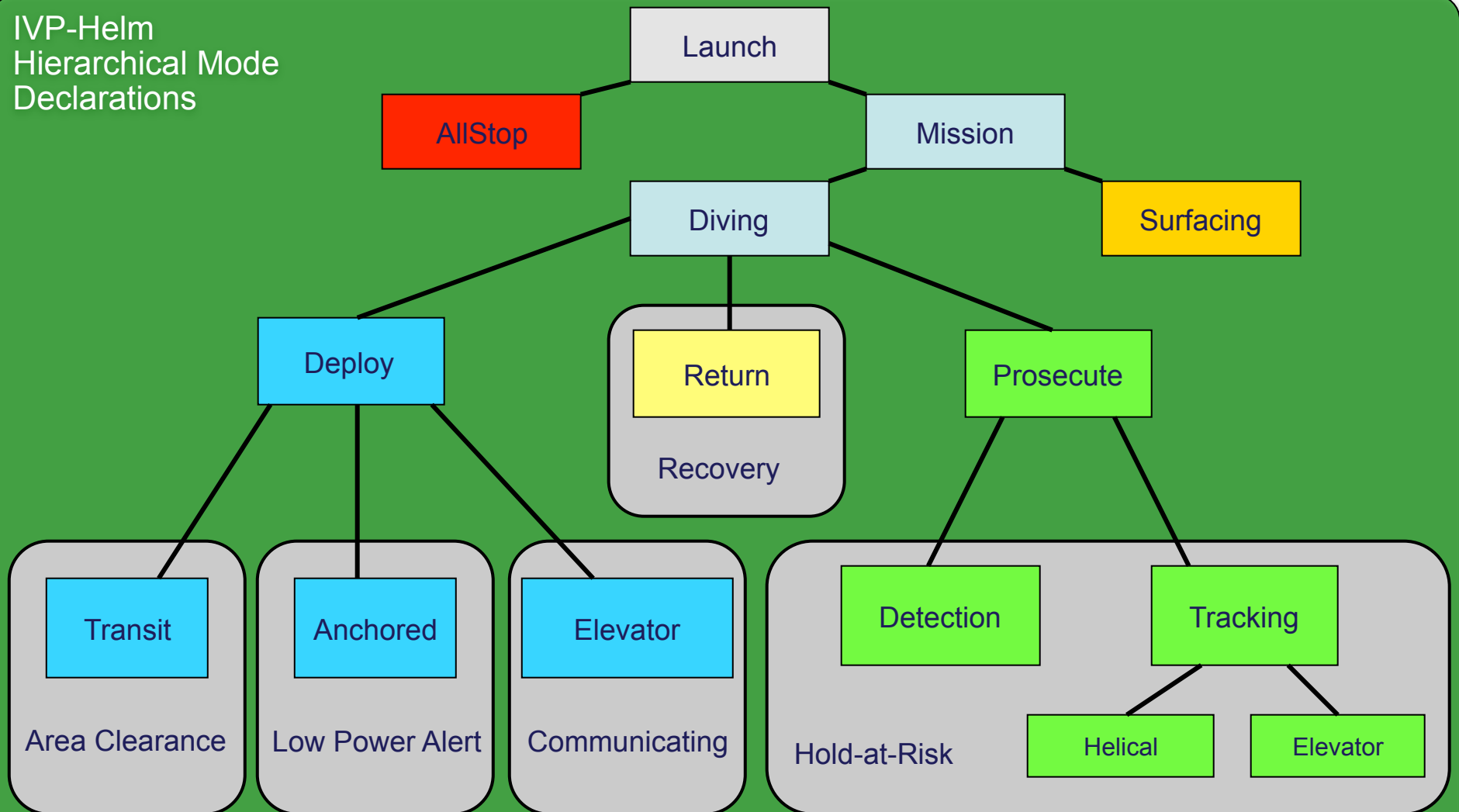
set MODE = LOITER_A {
  MODE = ACTIVE
  REGION = A
} LOITER_A
  
```





# MIT Prototype Autonomy Modes

IVP-Helm  
Hierarchical Mode  
Declarations



# Dynamic Modifications to the Helm

Q: How is the helm modified after launch?

A: By receipt of incoming MOOS mail.

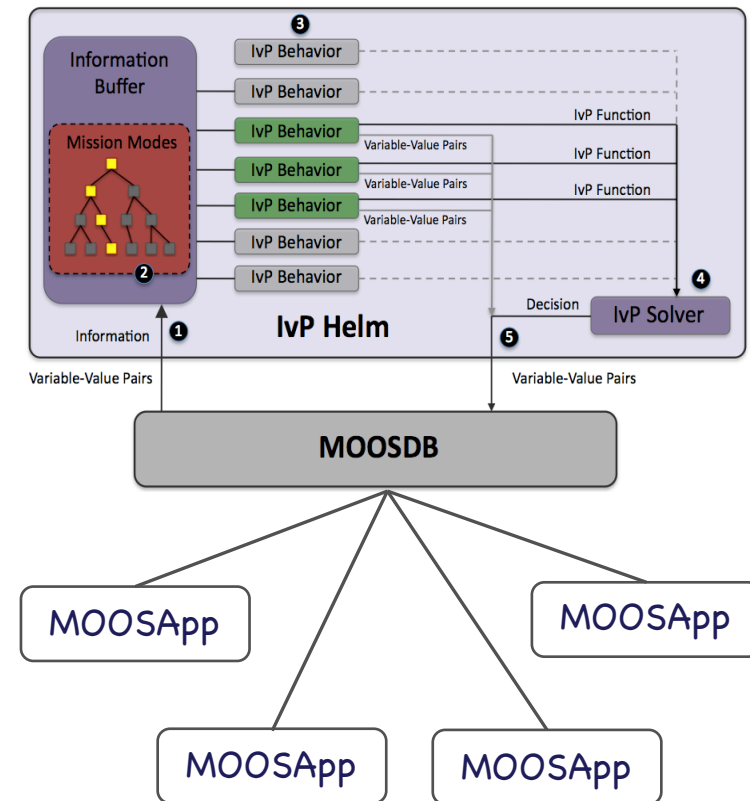
- The helm's mission mode may change
- Behavior parameters may change

Q: Why would the helm be modified?

- New internal plans generated
- Conclusions from sensor processing modules
- External comms from other vehicles
- External comms from field-control

Q: Why is this important?

A: It determines how the helm may interface with an off-board planner, field-control system, scheduler, and other vehicles.



# Dynamic Modifications to the Helm

Q: How is the helm modified after launch?

A: By receipt of incoming MOOS mail.

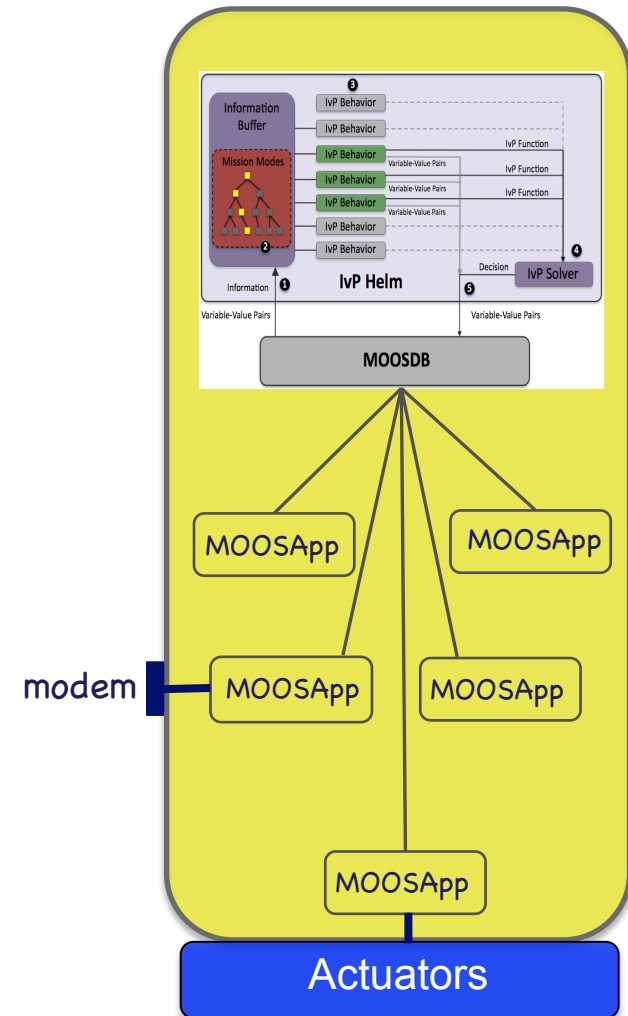
- The helm's mission mode may change
- Behavior parameters may change

Q: Why would the helm be modified?

- New internal plans generated
- Conclusions from sensor processing modules
- External comms from other vehicles
- External comms from field-control

Q: Why is this important?

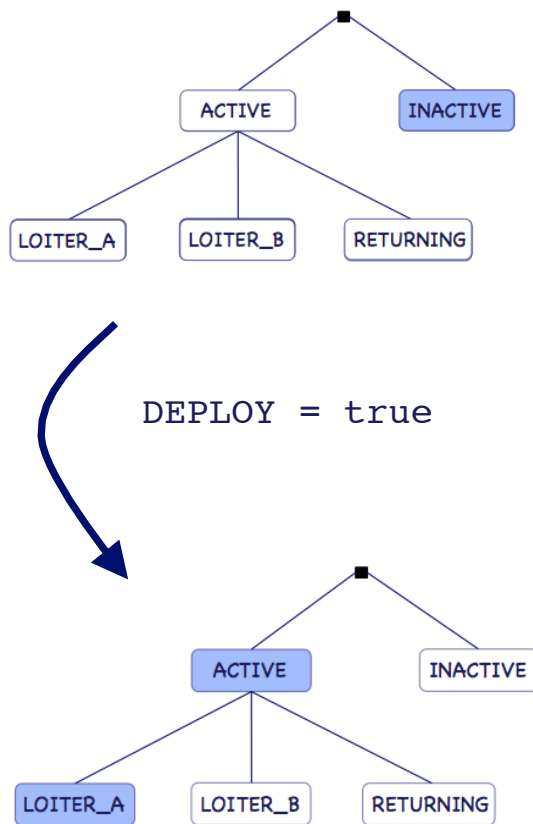
A: It determines how the helm may interface with an off-board planner, field-control system, scheduler, and other vehicles.



# Dynamic Modifications to the Helm

## (1) Mode Modifications

- The Helm Mode is determined by conditions
- The conditions are defined over MOOS variables:

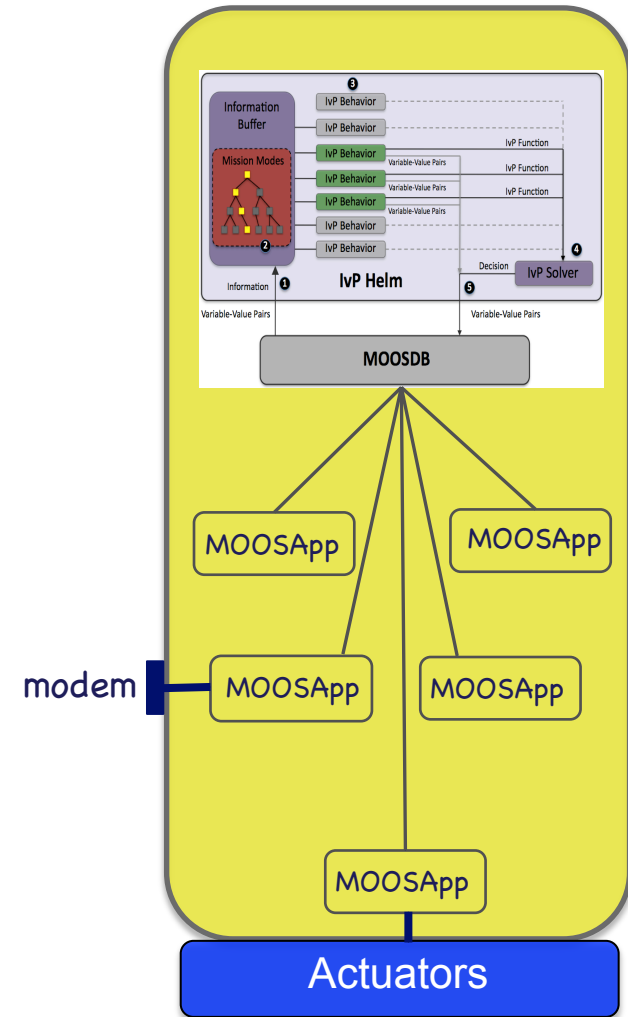


```

set MODE = ACTIVE {
  DEPLOY = true
} INACTIVE

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

set MODE = LOITER_A {
  MODE = ACTIVE
  REGION = A
} LOITER_A
  
```



# Dynamic Modifications to the Helm

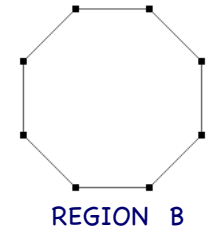
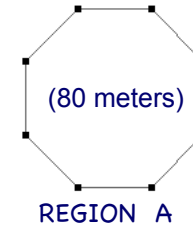
## (2) Behavior Modifications

- Behavior configuration parameters may altered after launch.
- Each behavior may specify an UPDATES variable.

```
Behavior = BHV_Loiter
{
  name      = loiter_a
  condition = (DEPLOY=true) and (REGION=A)

  speed = 1.8
  radius = 4.0
  polygon = format=radial,x=0,y=-75,radius=40,pts=8
  updates = LOITER_UPDATE
}
```

Launch and return position



LOITER\_UPDATE = **format=radial,x=160,y=-75,radius=40,pts=8**

```
Behavior = BHV_Loiter
{
  name      = loiter_a
  condition = (DEPLOY=true) and (REGION=A)

  speed = 1.8
  radius = 4.0
  format=radial,x=160,y=-75,radius=40,pts=8
  updates = LOITER_UPDATE
}
```



# Dynamic Behavior Spawning



## What is Dynamic Behavior Spawning?

- Behaviors may be defined as templates with instances spawned upon receipt of an externally generated, user-defined event.
- Behavior authors may implement behaviors to die under certain conditions, and post MOOS messages immediately prior to dying.

## Motivation:

- For certain behaviors, e.g., collision avoidance, contact tracking, multiple instances of the behavior are required, one for each contact.
- It's virtually impossible to know the amount or type of contacts encountered prior to the start of the mission.



# Configuring Behaviors with Dynamic Behavior Spawning

## Non-Templated Behavior:

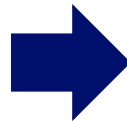
```
Behavior = BHV_AvoidCollision
{
  name      = avd_collision
  pwt       = 200
  condition = AVOID=true
  updates   = CONTACT_INFO ←

  contact = macrura
  active_outer_distance = 50
  active_inner_distance = 20
  completed_distance = 75
  collision_distance = 8
  all_clear_distance = 25
  active_grade = linear
  on_no_contact_ok = true
  extrapolate = true
  decay = 30,60
}
```

## Templated Behavior:

```
Behavior = BHV_AvoidCollision
{
  name      = avd_collision ←
  pwt       = 200
  condition = AVOID=true
  updates   = CONTACT_INFO ←
  endflag   = CONTACT_RESOLVED = ${CONTACT}
  templating = spawn

  contact = to-be-set
  active_outer_distance = 50
  active_inner_distance = 20
  completed_distance = 75
  collision_distance = 8
  all_clear_distance = 25
  active_grade = linear
  on_no_contact_ok = true
  extrapolate = true
  decay = 30,60
}
```



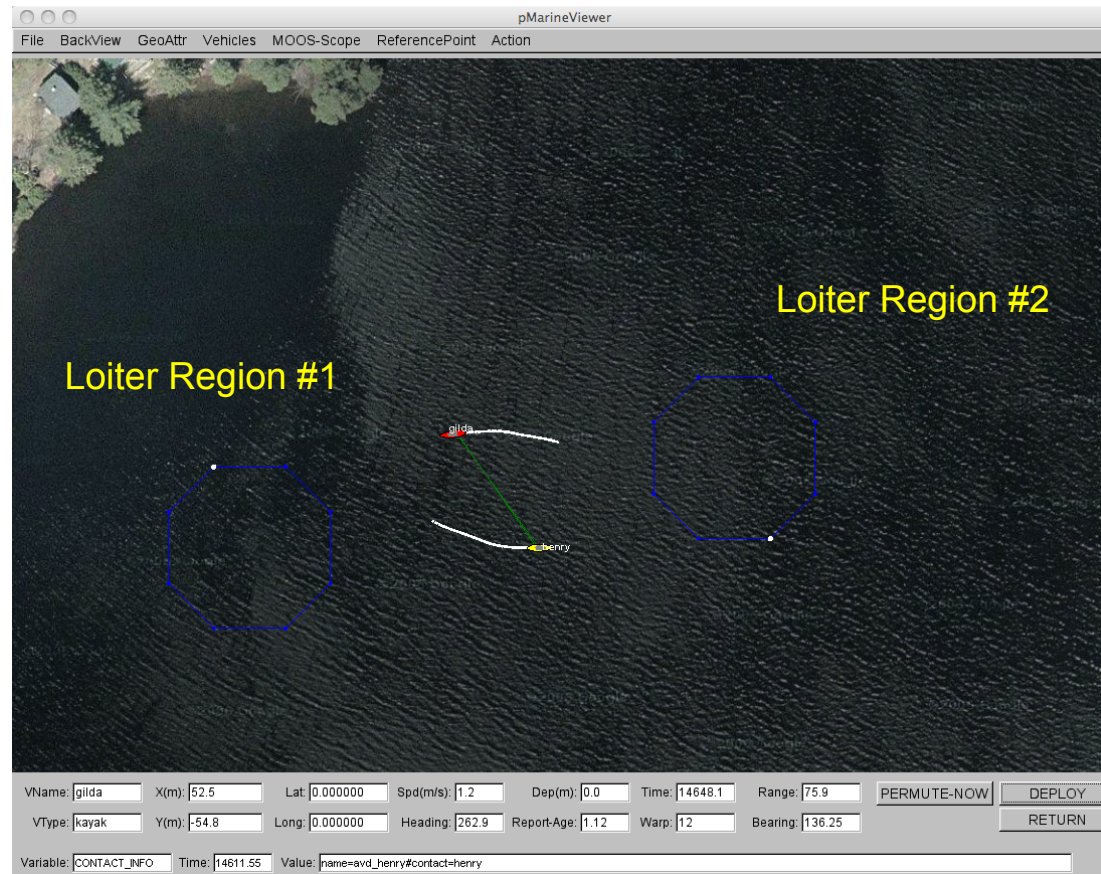
MOOS Post → CONTACT\_INFO = "name=avd\_macrura # contact=macrura"

MOOS Post → CONTACT\_INFO = "name=avd\_henry # contact=henry"

## The Berta Example Mission with Dynamic Behavior Spawning

### The Berta example mission:

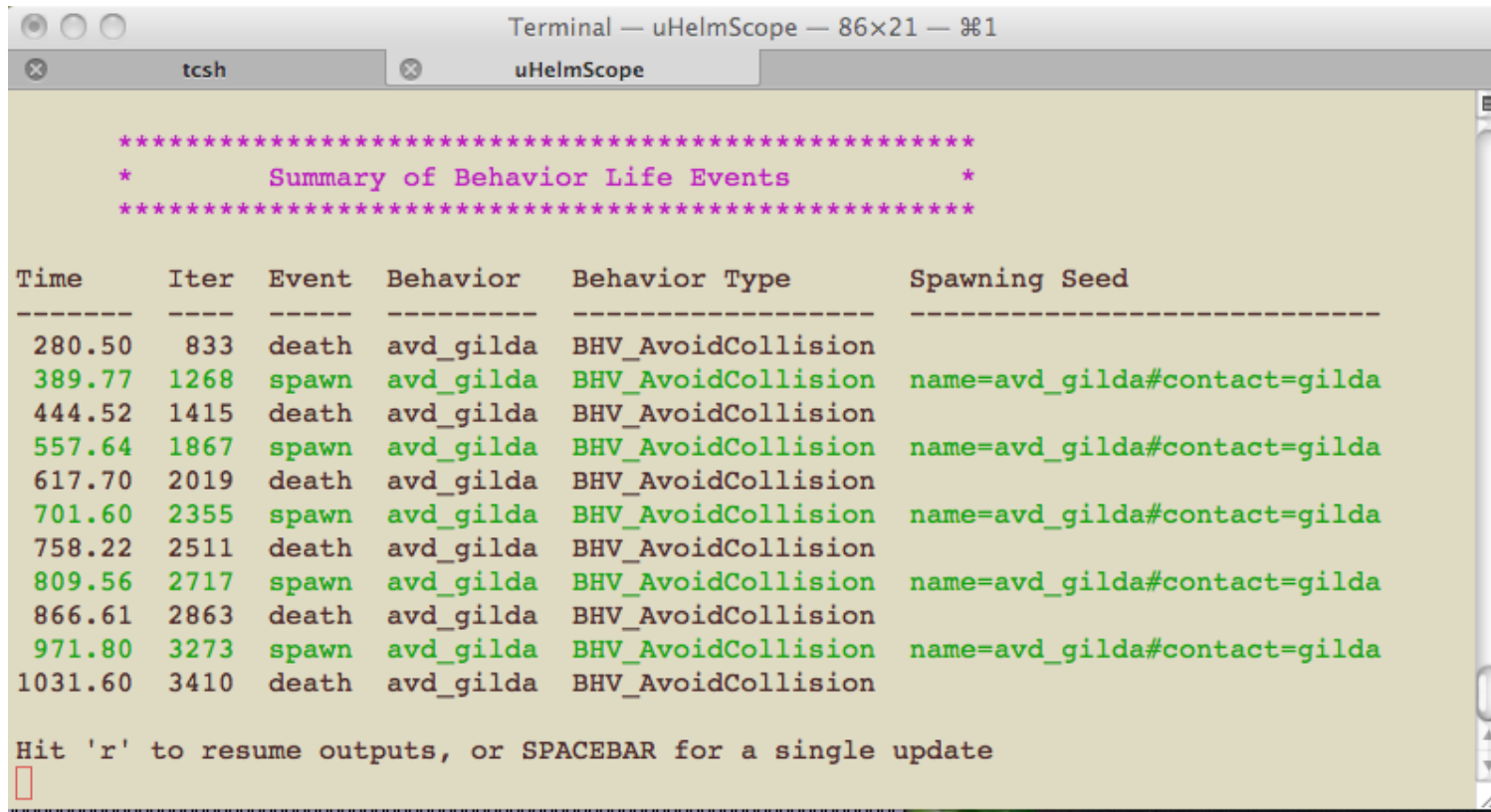
- In moos-ivp/trunk/missions/m2\_berta
- Two vehicles loitering and repeatedly swapping loiter locations
- Each time the vehicles get close, a collision avoidance behavior is spawned.
- After the range opens sufficiently, the collision avoidance behavior dies.



# Monitoring Life Events

- A “Life Event” is the *spawning or death* of a behavior.
- Life Events may be monitored in a special mode of the uHelmScope MOOS utility:

```
$ uHelmScope --life henry.moos
```



```

Terminal — uHelmScope — 86x21 — 81
tssh uHelmScope
*****
*           Summary of Behavior Life Events           *
*****

Time      Iter  Event  Behavior  Behavior Type  Spawning Seed
-----
280.50    833  death  avd_gilda  BHV_AvoidCollision
389.77    1268 spawn  avd_gilda  BHV_AvoidCollision  name=avd_gilda#contact=gilda
444.52    1415 death  avd_gilda  BHV_AvoidCollision
557.64    1867 spawn  avd_gilda  BHV_AvoidCollision  name=avd_gilda#contact=gilda
617.70    2019 death  avd_gilda  BHV_AvoidCollision
701.60    2355 spawn  avd_gilda  BHV_AvoidCollision  name=avd_gilda#contact=gilda
758.22    2511 death  avd_gilda  BHV_AvoidCollision
809.56    2717 spawn  avd_gilda  BHV_AvoidCollision  name=avd_gilda#contact=gilda
866.61    2863 death  avd_gilda  BHV_AvoidCollision
971.80    3273 spawn  avd_gilda  BHV_AvoidCollision  name=avd_gilda#contact=gilda
1031.60   3410 death  avd_gilda  BHV_AvoidCollision

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

```

# Analyzing Life Events

- A “Life Event” is the *spawning* or *death* of a behavior.
- Life Events may be monitored in a special mode of the uHelmScope MOOS utility.
- The Life Event may also be examined post-runtime from the MOOS log files:

```
$ aloghelm --life henry_logfile.alog
```

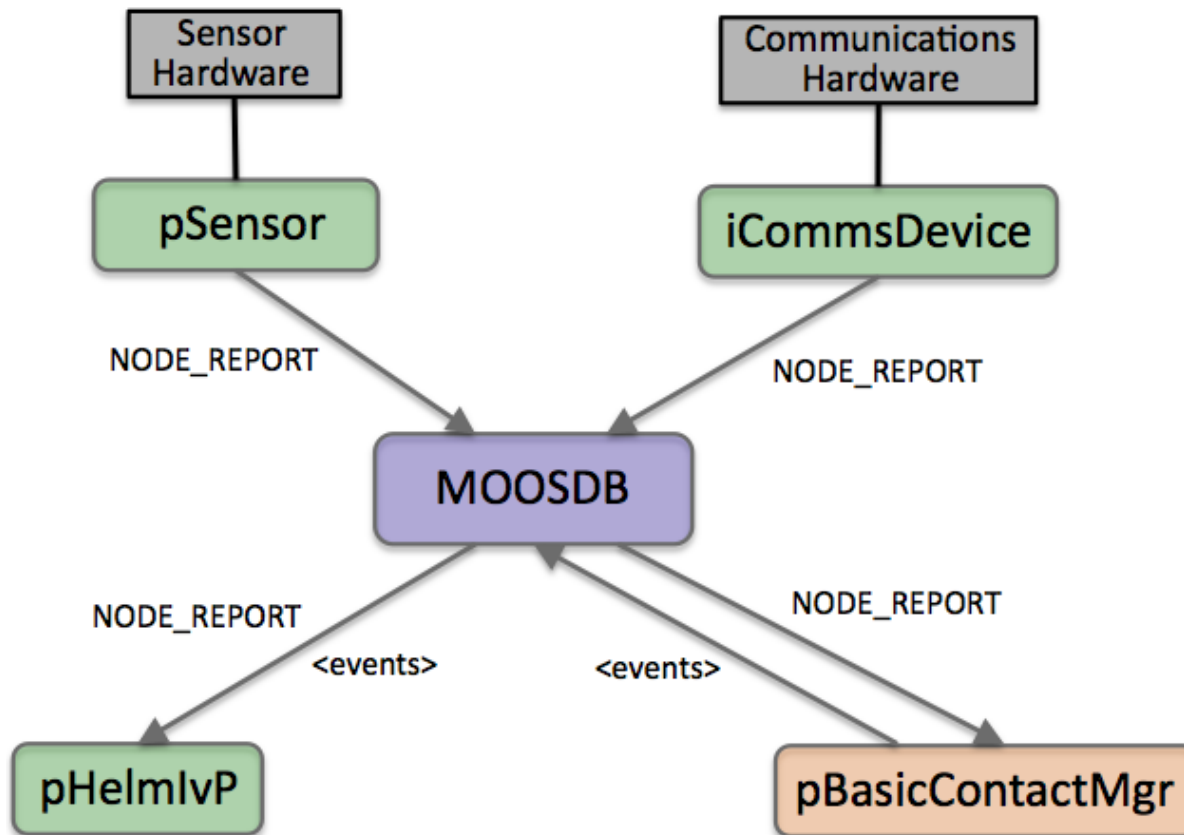
```
Terminal - tcsh - 94x31 - 93
sh pMarinePID tcsh

*****
* Summary of Behavior Life Events *
*****

Time      Iter  Event  Behavior      Behavior Type      Spawning Seed
-----
0.00      1  spawn  loiter        BHV_Loiter         helm startup
0.00      1  spawn  waypt_return  BHV_Waypoint       helm startup
0.00      1  spawn  station-keep  BHV_StationKeep    helm startup
94.57    247  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
152.94   398  death  avd_gilda     BHV_AvoidCollision
222.32   677  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
280.50   833  death  avd_gilda     BHV_AvoidCollision
389.77  1268  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
444.52  1415  death  avd_gilda     BHV_AvoidCollision
557.64  1867  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
617.70  2019  death  avd_gilda     BHV_AvoidCollision
701.60  2355  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
758.22  2511  death  avd_gilda     BHV_AvoidCollision
809.56  2717  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
866.61  2863  death  avd_gilda     BHV_AvoidCollision
971.80  3273  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
1031.60 3410  death  avd_gilda     BHV_AvoidCollision
1164.46 3927  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
1226.22 4061  death  avd_gilda     BHV_AvoidCollision
1341.33 4503  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
1400.98 4638  death  avd_gilda     BHV_AvoidCollision
1520.36 5108  spawn  avd_gilda     BHV_AvoidCollision name=avd_gilda#contact=gilda
leonardo:missions/m2_berta(trunk)%
```

# Contact Management and Behavior Spawning

- A MOOS application - pBasicContactMgr.
- It receives NODE\_REPORT messages from other MOOS applications





# Non-Traditional Aspects of Behavior-Based Control in the IvP Helm



- Behaviors have state.
- Behaviors influence each other between iterations.
- Behaviors accept externally generated plans.
- There may be several instances of the same behavior.
- Behaviors may spawn and die dynamically based on events or commands.
- Behaviors may run in a configurable sequence.
- Behaviors rate actions over a coupled decision space (multi-objective optimization)



This is not Rodney Brooks' Behavior Based Control

However - the power of independent, incremental development has been retained, enhanced by the power of Open Source development, and wide, diverse collaborations.

# The Waypoint Behavior

## (General Characteristics)

Purpose: Traverse a given set of waypoints, gracefully handling missed vertices.

Parameters:

- points*: A set of points in the X-Y plane
- capture\_radius*: Distance from a point, within which arrival is declared.
- speed*: Desired speed of traversal.
- non-monotonic\_radius*: Distance from a point, within which an increase in distance is treated as an arrival.
- track\_lead*: For track-line following - distance to perpendicular intersection point.
- order*: Order of point traversal.
- repeat*: Number of times points are traversed.

Example:

- points*: (0,-80), (45,-45), (160, -120)
- capture\_radius*: 4
- speed*: 2.5



# The Waypoint Behavior

## (Non-monotonic Radius)

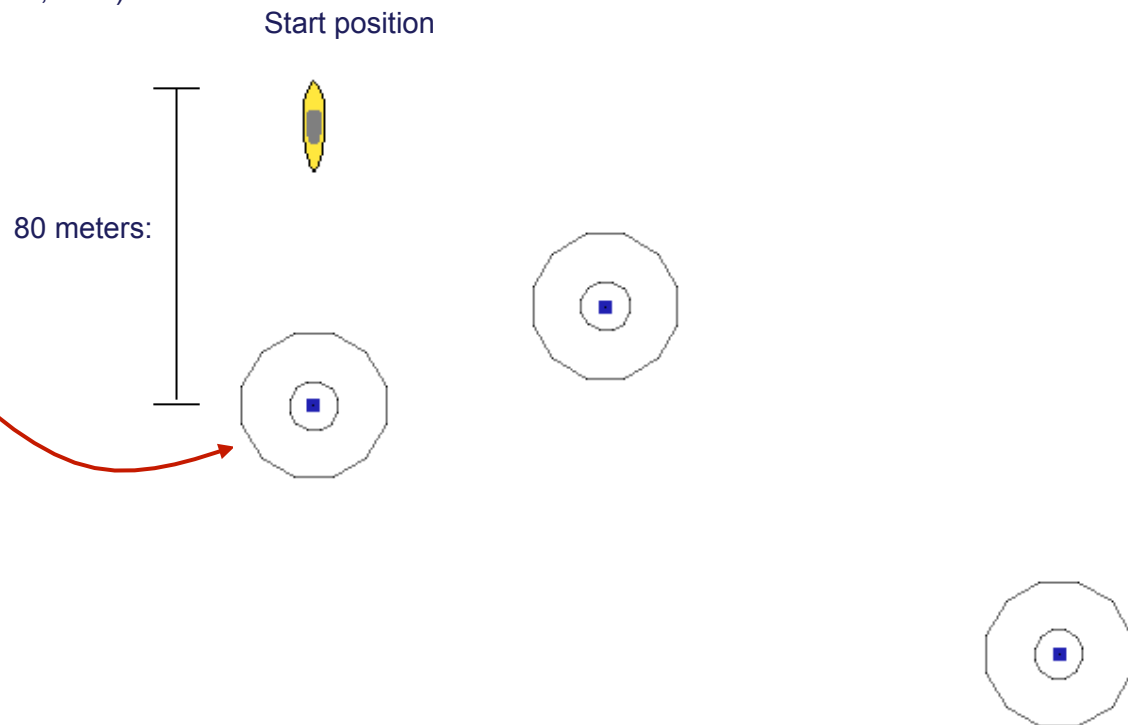
Purpose: Traverse a given set of waypoints, gracefully handling missed vertices.

Parameters:

- points*: A set of points in the X-Y plane
- capture\_radius*: Distance from a point, within which arrival is declared.
- speed*: Desired speed of traversal.
- non-monotonic\_radius*: Distance from a point, within which an increase in distance is treated as an arrival.
- track\_lead*: For track-line following - distance to perpendicular intersection point.
- order*: Order of point traversal.
- repeat*: Number of times points are traversed.

Example:

- points*: (0,-80), (45,-45), (160, -120)
- capture\_radius*: 4
- speed*: 2.5
- non-monotonic\_radius*: 12





# The Waypoint Behavior

## (Track-line Following)

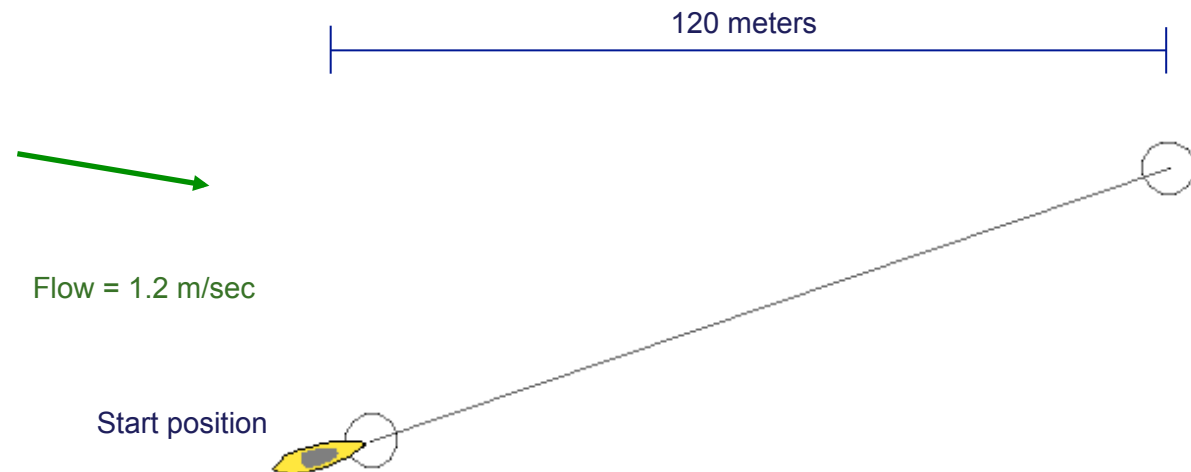
Purpose: Traverse a given set of waypoints, gracefully handling missed vertices.

Parameters:

*points*: A set of points in the X-Y plane  
*capture\_radius*: Distance from a point, within which arrival is declared.  
*speed*: Desired speed of traversal.  
*non-monotonic\_radius*: Distance from a point, within which an increase in distance is treated as an arrival.  
*track\_lead*: For track-line following - distance to perpendicular intersection point.  
*order*: Order of point traversal.  
*repeat*: Number of times points are traversed.

Example:

*points*: (0,-45), (120,0)  
*capture\_radius*: 4  
*speed*: 2.0  
*track\_lead*: 0



# The Waypoint Behavior

## (Track-line Following)

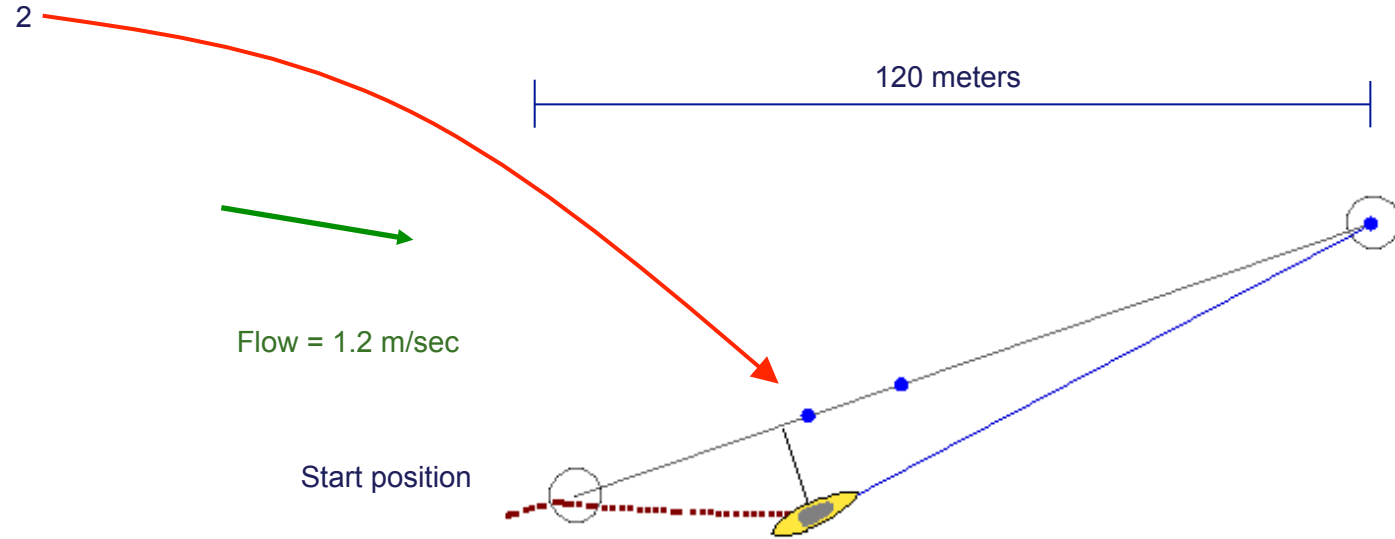
Purpose: Traverse a given set of waypoints, gracefully handling missed vertices.

Parameters:

- points*: A set of points in the X-Y plane
- capture\_radius*: Distance from a point, within which arrival is declared.
- speed*: Desired speed of traversal.
- non-monotonic\_radius*: Distance from a point, within which an increase in distance is treated as an arrival.
- track\_lead*: For track-line following - distance to perpendicular intersection point.
- order*: Order of point traversal.
- repeat*: Number of times points are traversed.

Example:

- points*: (0,-45), (120,0)
- capture\_radius*: 4
- speed*: 2.0
- track\_lead*: 2



# The Obstacle Avoidance Behavior

Purpose: Avoid a set of given obstacles each represented by a *convex* polygon.

Parameters:

- polygon*: A set of points in the X-Y plane, comprising a convex polygon.
- polygon*: Other obstacles.
- allowable\_ttc*: Time To Collision Allowed before a candidate maneuver is penalized.
- activation\_dist*: Distance to a polygon beyond which the behavior is inactive.
- buffer\_dist*: Distance to polygon treated as a collision.

Example:

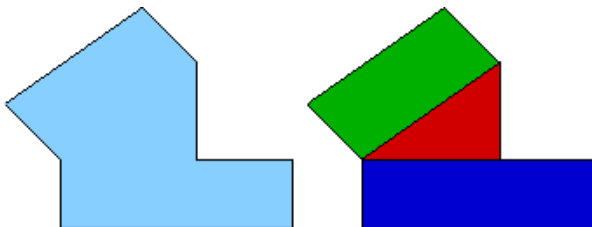
- polygon*: (10,10), (20,20), (30,30), (40,40)
- polygon*: (60,60), (70,70), (80,80), (90,90)
- allowable\_ttc*: 25
- activation\_dist*: 60
- buffer\_dist*: 8

Q: Why convex polygons?

A: Most operations are simplified.

- Point containment test,
- Distance to polygon,
- Line segment intersection, etc.

A: A non-convex polygon can be represented by a set of convex polygons.



Start



Waypoint



# The Obstacle Avoidance Behavior

Purpose: Avoid a set of given obstacles each represented by a *convex* polygon.

Parameters:

- polygon*: A set of points in the X-Y plane, comprising a convex polygon.
- allowable\_ttc*: Time To Collision Allowed before a candidate maneuver is penalized.
- activation\_dist*: Distance to a polygon beyond which the behavior is inactive.
- buffer\_dist*: Distance to polygon treated as a collision.

Example:

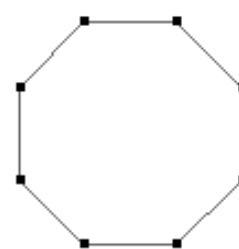
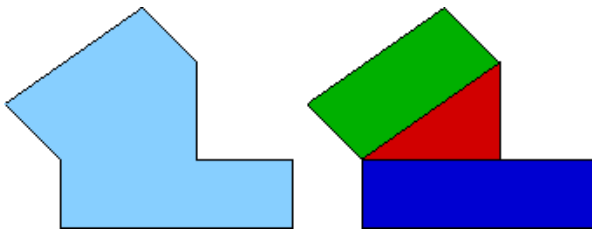
- polygon*: (10,10), (20,20), (30,30), (40,40)
- polygon*: (60,60), (70,70), (80,80), (90,90)
- allowable\_ttc*: 25
- activation\_dist*: 60
- buffer\_dist*: 8

Q: Why convex polygons?

A: Most operations are simplified.

- Point containment test,
- Distance to polygon,
- Line segment intersection, etc.

A: A non-convex polygon can be represented by a set of convex polygons.

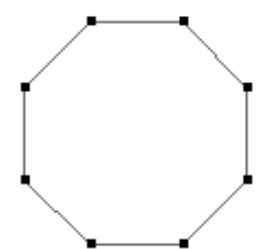


REGION A

Polygon #1



Polygon #2



REGION B



# The Loiter Behavior

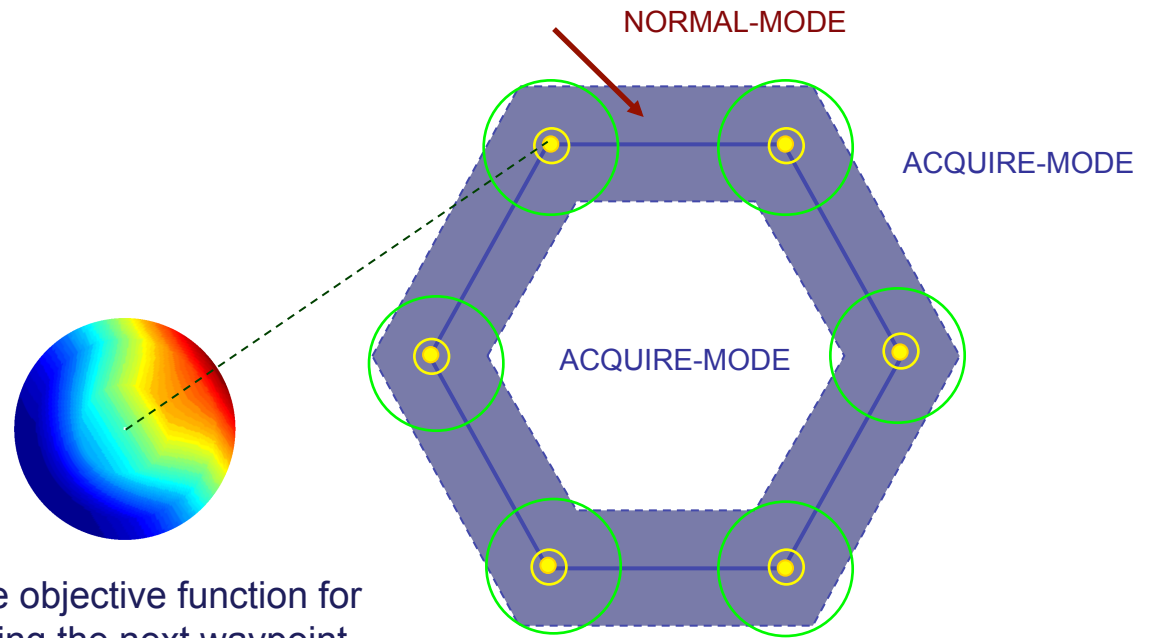
Purpose: Repeatedly traverse a given set of waypoints, gracefully handling missed vertices.  
Automatically calculate trajectory re-entry when required.

Parameters:

- polygon*: A set of points in the X-Y plane, comprising a convex polygon.
- capture\_radius*: Distance from a point, within which arrival is declared.
- speed*: Desired speed of traversal.
- non-monotonic\_radius*: Distance from a point, within which an increase in distance is treated as an arrival.
- acquire\_distance*: Distance from the polygon, outside of which the behavior is in "acquire mode".
- clockwise*: True if traversing clockwise.

Example:

- polygon*: radial:50,60,40,6
- capture\_radius*: 10
- non-monotonic\_radius*: 15
- acquire\_distance*: 15
- clockwise*: true



Vehicle objective function for achieving the next waypoint

# The Loiter Behavior

## (Acquire Vertex Policy - External Case)

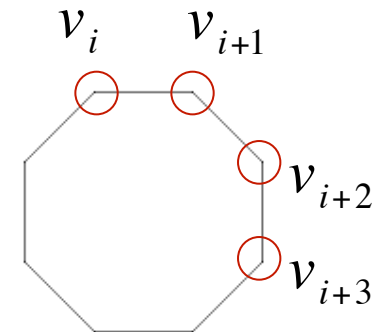
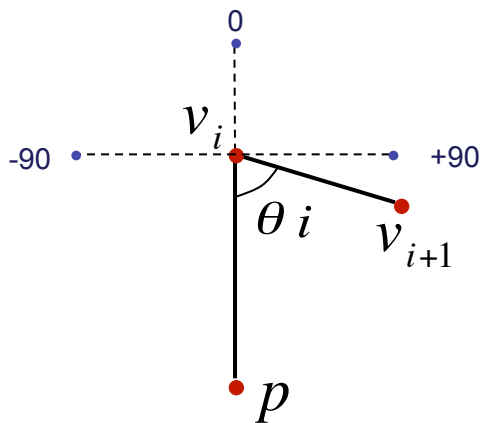
Purpose: Repeatedly traverse a given set of waypoints, gracefully handling missed vertices.  
 Automatically calculate trajectory re-entry when required.

Parameters:

- polygon*: A set of points in the X-Y plane, comprising a convex polygon.
- capture\_radius*: Distance from a point, within which arrival is declared.
- speed*: Desired speed of traversal.
- non-monotonic\_radius*: Distance from a point, within which an increase in distance is treated as an arrival.
- acquire\_distance*: Distance from the polygon, outside of which the behavior is in "acquire mode".
- clockwise*: True if traversing clockwise.

### Acquire Vertex Policy (External Case):

acquire\_vertex =  $v_i$   
 where  $i = \text{argmin}(\theta_i)$   
 $v_i$  is viewable from  $p$



200 meters



# The Loiter Behavior

## (Acquire Vertex Policy - Internal Case)

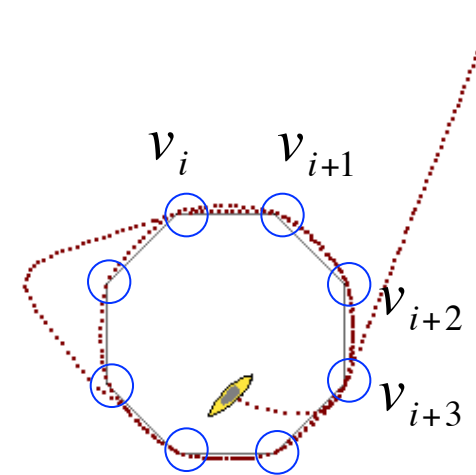
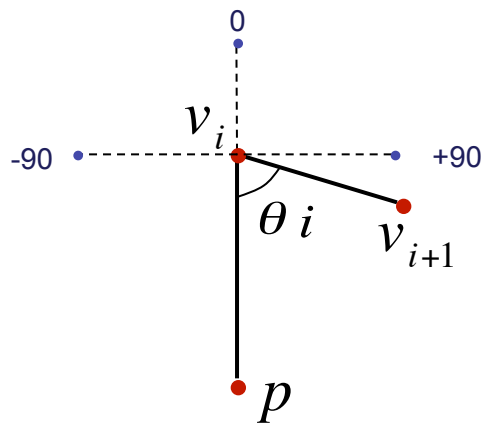
Purpose: Repeatedly traverse a given set of waypoints, gracefully handling missed vertices.  
Automatically calculate trajectory re-entry when required.

Parameters:

- polygon*: A set of points in the X-Y plane, comprising a convex polygon.
- capture\_radius*: Distance from a point, within which arrival is declared.
- speed*: Desired speed of traversal.
- non-monotonic\_radius*: Distance from a point, within which an increase in distance is treated as an arrival.
- acquire\_distance*: Distance from the polygon, outside of which the behavior is in "acquire mode".
- clockwise*: True if traversing clockwise.

### Acquire Vertex Policy (Internal Case):

acquire\_vertex =  $v_i$   
 where  $i = \operatorname{argmin}(\theta_j + k_j)$   
 $k_j = -c$  if  $v_j$  is current  
 0 otherwise



200 meters



# The Turn-Limit Behavior

Purpose: To limit the rate of vehicle turn to protect a towed sensor.

Parameters:

*memory\_time*: A set of points in the X-Y plane, comprising a convex polygon.

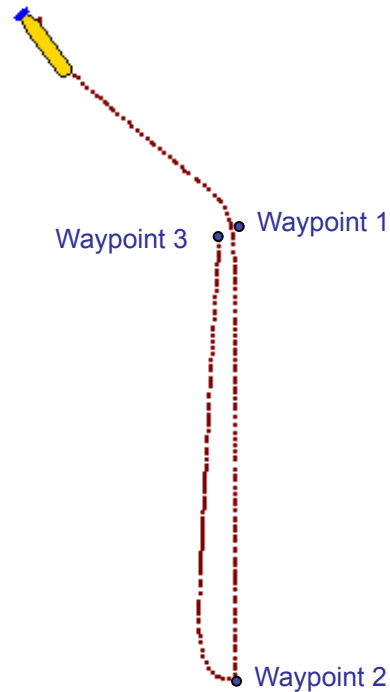
*turn\_range*: Distance from a point, within which arrival is declared.

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

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

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

$$\text{turn radius: } r = v / ((u/180)\pi)$$





# The Turn-Limit Behavior

Purpose: To limit the rate of vehicle turn to protect a towed sensor.

Parameters:

*memory\_time*: A set of points in the X-Y plane, comprising a convex polygon.

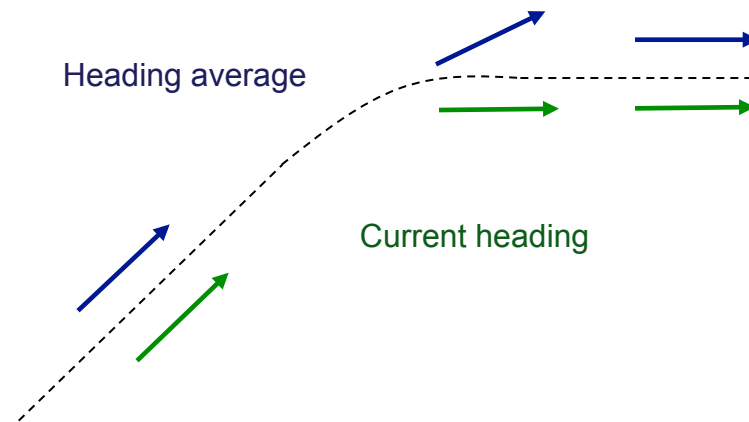
*turn\_range*: Distance from a point, within which arrival is declared.

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

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

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

turn radius:  $r = v / ((u/180)\pi)$



# The Turn-Limit Behavior

Purpose: To limit the rate of vehicle turn to protect a towed sensor.

Parameters:

*memory\_time*: A set of points in the X-Y plane, comprising a convex polygon.

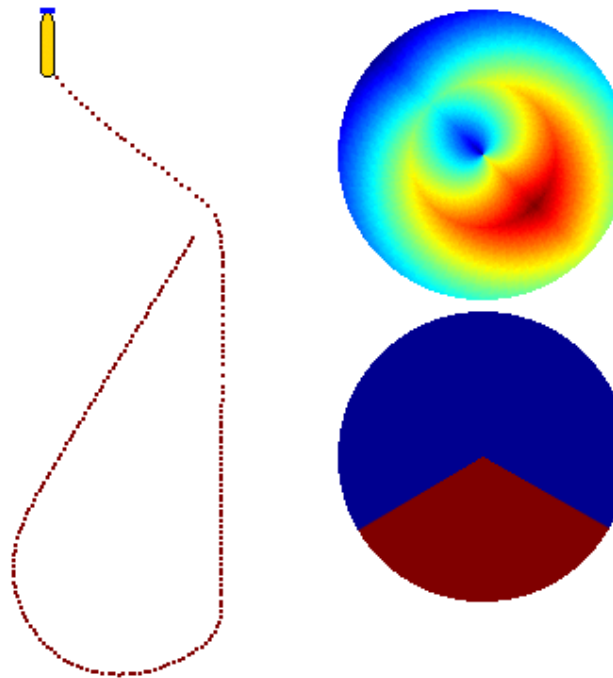
*turn\_range*: Distance from a point, within which arrival is declared.

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

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

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

turn radius:  $r = v / ((u/180)\pi)$



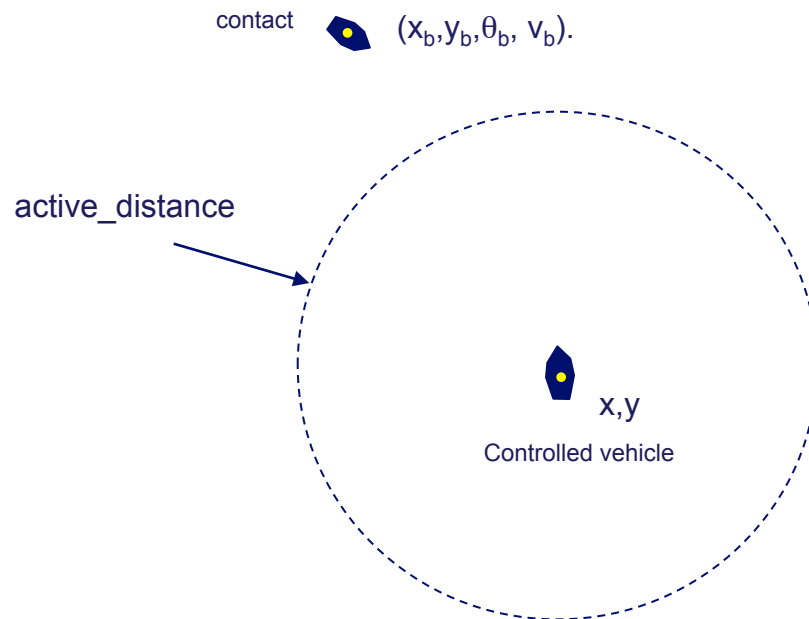
# The Collision Avoidance Behavior

Purpose: Avoid collision with a vehicle with given position and trajectory.

Parameters:

- contact*: Contact ID of the vehicle to be avoided.
- active\_distance*: Distance to the contact beyond which the behavior has no influence on autonomy
- all\_clear\_distance*: CPA Distance to the contact below which penalty begins
- collision\_distance*: CPA Distance to the contact at which penalty is maximize (treated as a collision).

Utility based on CPA (closest point of approach)



Distance given by Pythagorean theorem in known trajectories.  
Express distance a function of angle, speed and time-on-leg.

$$\gamma^2(\theta, v, t) = k_2 t^2 + k_1 t + k_0$$

$$k_2 = \cos(\theta)v^2 - 2\cos(\theta)v\cos(\theta_b)v_b + \cos^2(\theta_b)v_b^2 + \sin^2(\theta)v^2 - 2\sin(\theta)v\sin(\theta_b)v_b + \sin^2(\theta_b)v_b^2$$

$$k_1 = 2\cos(\theta)vy - 2\cos(\theta)vy_b - 2y\cos(\theta_b)v_b + 2\cos(\theta_b)v_b y_b + 2\sin(\theta)vx - 2\sin(\theta)vx_b - \sin(\theta_b)xv_b + 2\sin(\theta_b)v_b x_b$$

$$k_0 = y^2 - 2yy_b + y_b^2 - 2xx_b + x_b^2$$

First derivative w.r.t. time has only one root:

$$\gamma^2(\theta, v, t)' = 2k_2 t + k_1$$

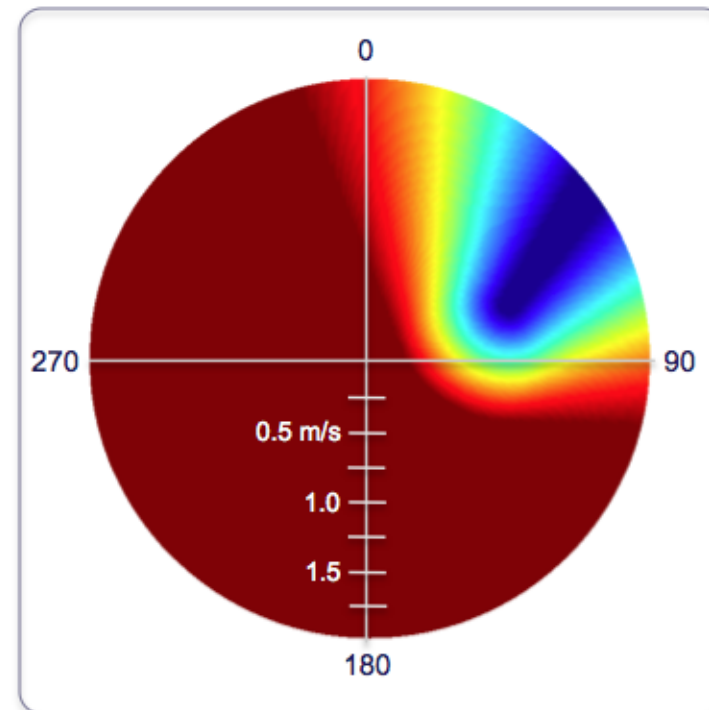
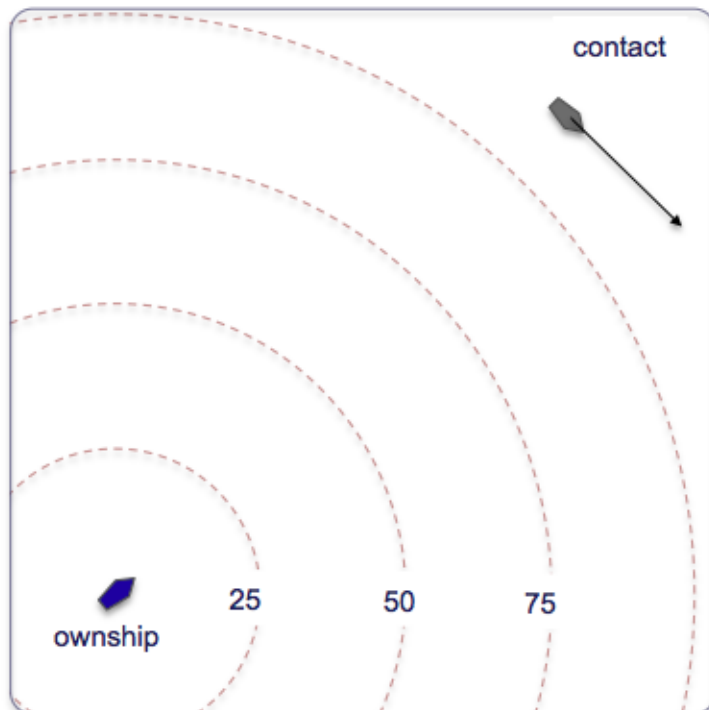
$$t = -\frac{k_1}{2k_2}$$

# The Collision Avoidance Behavior

Purpose: Avoid collision with a vehicle with given position and trajectory.

Parameters:

- contact*: Contact ID of the vehicle to be avoided.
- active\_distance*: Distance to the contact beyond which the behavior has no influence on autonomy
- all\_clear\_distance*: CPA Distance to the contact below which penalty begins
- collision\_distance*: CPA Distance to the contact at which penalty is maximize (treated as a collision).



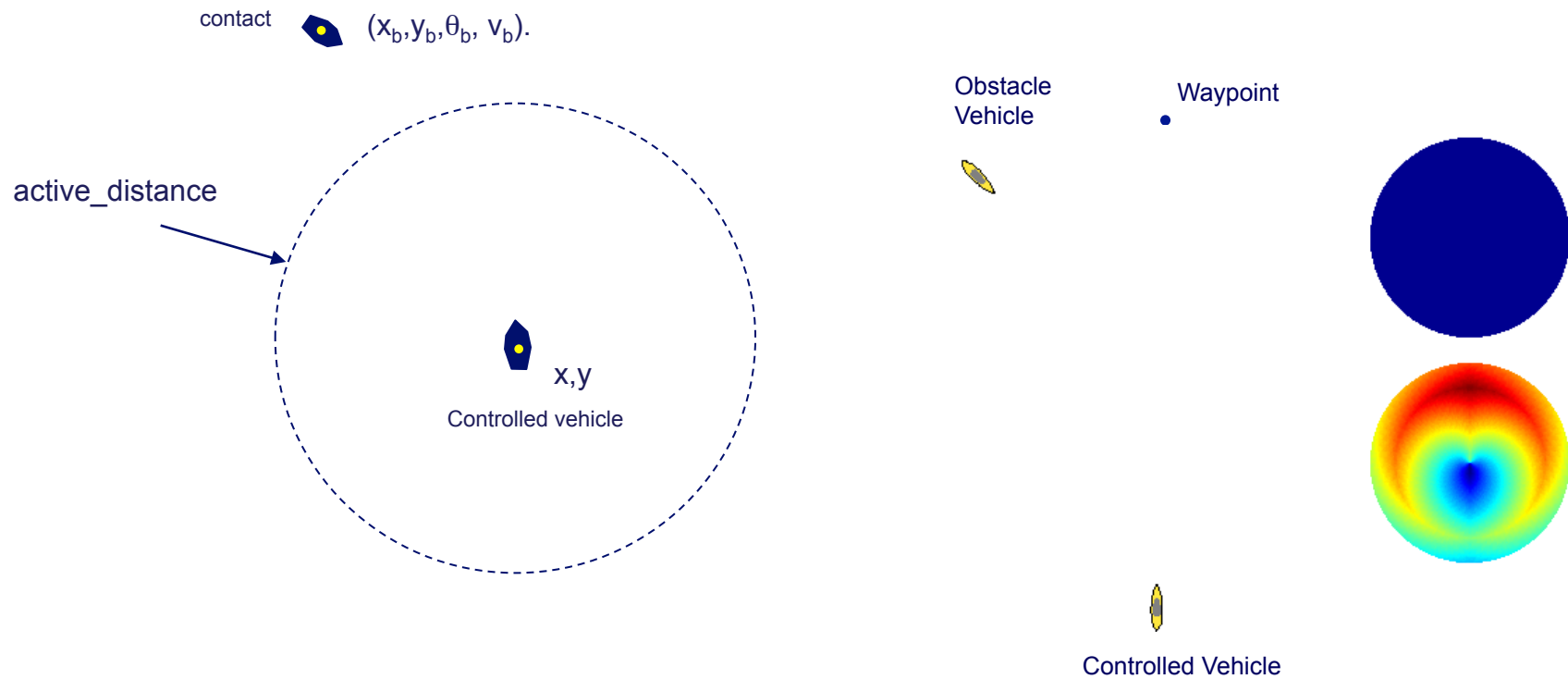
# The Collision Avoidance Behavior

Purpose: Avoid collision with a vehicle with given position and trajectory.

Parameters:

- contact*: Contact ID of the vehicle to be avoided.
- active\_distance*: Distance to the contact beyond which the behavior has no influence on autonomy
- all\_clear\_distance*: CPA Distance to the contact below which penalty begins
- collision\_distance*: CPA Distance to the contact at which penalty is maximize (treated as a collision).

Utility based on CPA (closest point of approach)





# Outline



- Trends in autonomous marine vehicles
- The Payload Autonomy Paradigm and the MOOS-IvP project
- Multi-Objective Optimization with Interval Programming
- The IvP Helm
- MOOS-IvP 4.2 and Plans for Future Development



# MOOS-IvP 4.2



## Changes to the Helm

- Preparation for multiple IvP Functions per behavior.
- Preparations for IvP Function re-use.
- Journaling of IVPHELM\_STATUS messages (to reduce log file output)

## New MOOS Apps

- uSimBeaconRange
- uSimActiveSonar
- uSimCurrent

## Improvements to Existing MOOS Apps

- pMarineViewer
- uSimMarine
- alogview
- uFunctionVis



# MOOS-IvP 4.2



Grab File Edit Capture Window Help Stop Recording (100%) Wed 11:13 AM

pMarineViewer

File BackView GeoAttr Vehicles MOOS-Scope Action

betty's next waypoint

betty (SURVEYING)

betty waypoint

PNA (State Report: 836)

VName: <input type="text" value="betty"/>	X(m): <input type="text" value="107.7"/>	Lat: <input type="text" value="42.357376"/>	Spd(m/s): <input type="text" value="1.3"/>	Dep(m): <input type="text" value="0.0"/>	Time: <input type="text" value="1939.0"/>	<input type="button" value="STATION"/>	<input type="button" value="DEPLOY"/>
VType: <input type="text" value="kayak"/>	Y(m): <input type="text" value="-124.4"/>	Long: <input type="text" value="-71.086267"/>	Heading: <input type="text" value="105.2"/>	Report-Age: <input type="text" value="1.27"/>	Warp: <input type="text" value="1"/>	<input type="button" value="RETURN"/>	

Variable:  Time:  Value:



## MONTEREY CALIFORNIA AUGUST 2006

### Notable "First's":

- First usage of the payload autonomy paradigm
- First significant usage of MOOS-IvP on a UUV
- First deployment of a UUV with a vector sensor array

